

USENIX

C++ CONFERENCE PROCEEDINGS

USENIX

C++

CONFERENCE  
PROCEEDINGS

Portland, Oregon  
August 10 - 13, 1992

SUMMER

1992

For additional copies of these proceedings contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA

The price is \$30 for members and \$39 for nonmembers.

Outside the U.S.A and Canada, please add  
\$20 per copy for postage (via air printed matter).

Past USENIX C++ Proceedings

C++ Conference	April 1991	Washington, DC	\$22/26
C++ Conference	April 1990	San Francisco, CA	\$28
C++ Conference	October 1988	Denver, CO	\$30
C++ Workshop	November 1987	Santa Fe, NM	\$30

Outside the U.S.A and Canada, please add  
\$20 per copy for postage (via air printed matter).

Copyright © 1992 by The USENIX Association  
All rights reserved.

ISBN 1 -880446-45-6

This volume is published as a collective work.  
Rights to individual papers remain  
with the author or the author's employer.

UNIX is a registered trademark of UNIX System Laboratories.  
Other trademarks are noted in the text.

Printed in the United States of America on 50% recycled paper, 10-15% post-consumer waste.





**USENIX C++ Technical Conference**  
**Proceedings**

**USENIX Association**

**August 10 - 13, 1992**  
**Portland, Oregon**

## Program Committee

Jonathan E. Shapiro, *UNIX System Laboratories, Inc. (Chair)*

Dag M. Brück, *Lund Institute of Technology, Sweden*

Theodore C. Goldstein, *Sun Microsystems Laboratories*

Keith Gorlen, *National Institutes of Health*

Brian M. Kennedy, *Intellection, Inc.*

Dmitry Lenkov, *Hewlett Packard*

Mark Linton, *Silicon Graphics*

Scott Meyers, *Brown University*

Barbara E. Moo, *AT&T Bell Laboratories*

Martin O'Riordan, *Microsoft*

Jim Waldo, *Sun Microsystems, Inc.*

# Program and Table of Contents

## USENIX C++ Technical Conference

August 10-13, 1992

Portland, Oregon

### Wednesday, August 12

**Keynote Address: 9:00 - 10:00 a.m.**

The Essentials of Object-Oriented Programming

*Kristen Nygaard, Department of Informatics, University of Oslo*

**Session 1: 10:30 - 12:30**

*Chair: Doug Lea, SUNY Oswego*

Smart Pointers: They're Smart, But They're Not Pointers.....1

*Daniel R. Edelson, INRIA Project SOR*

Not a Language Extension.....21

*Martin D. Carroll, AT&T Bell Laboratories*

Garbage Collection and Run-time Typing as a C++ Library.....37

*David Detlefs, Digital Equipment Corporation*

Encapsulating a C++ Library.....57

*Mark Linton, Silicon Graphics, Inc.*

**Session 2 2:00 - 3:30**

*Chair: Jim Waldo, Sun Microsystems*

Sniff: A Pragmatic Approach to a C++ Programming Environment.....67

*Walter R. Bischofberger, Union Bank of Switzerland*

A Statically Typed Abstract Representation for C++ Programs.....83

*Robert B. Murray, AT&T Bell Laboratories*

CCEL: A Metalanguage for C++.....99

*Carolyn K. Duby, Scott Meyers, Steven P. Reiss, Brown University*

**Session 3: 4:00 - 5:30**

*Chair: Theodore C. Goldstein, Sun Microsystems*

Space-Efficient Trees in C++.....117

*Andrew Koenig, AT&T Bell Laboratories*

High-Performance Scientific Computing Using C++.....131

*K. G. Budge, J. S. Perry, A. C. Robinson, Sandia National Laboratories*

O-R Gateway: A System for Connecting C++ Application Programs and  
Relational Databases.....151

*Abdullah Alashqur, Craig Thompson, Texas Instruments*

## Thursday, August 13

### Session 4: 9:00 - 10:30 a.m.

Chair: Keith Gorlen, NIH

Static Initializers: Reducing the Value-Added Tax on Programs.....171  
*John F. Reiser, Mentor Graphics Corporation*

Cdiff: A Syntax Directed Differencer for C++ Programs.....181  
*Judith E. Gross, AT&T Bell Laboratories*

C++ in a Changing Environment.....195  
*Andrew J. Palay, Silicon Graphics Computer Systems*

### Session 5: 11:00 - 12:30

Chair: Dag Brück, Lund Institute

Adding Concurrency to a Programming Language.....207  
*Peter A. Buhr, Glen Ditchfield, University of Waterloo*

A Portable Implementation of C++ Exception Handling.....225  
*Don Cameron, Paul Faust, Dmitry Lenkov, Mickey Mehta,  
Hewlett-Packard California Language Laboratory*

An Assertion Mechanism Based on Exceptions.....245  
*Philippe Gautron, Université Paris VI, LITP-IBP*

### Session 6: 2:00 - 3:30

Chair: Susan E. Waggoner, US WEST

A Communication Facility for Distributed Object-Oriented Applications.....263  
*Afshin Daghi, Pierre Delisle, Salil Deshpande, Sun Microsystems, Inc.*

Writing a Client-Server Application in C++.....279  
*Paulo Guedes, Open Software Foundation*

Integrating the Sun Microsystems XDR/RPC Protocols into the C++ Stream Model.....295  
*Robert E. Minnear, Patrick A. Muckelbauer, Vincent F. Russo, Purdue University*

### Session 7: 4:00 - 5:30

Chair: Mark Linton, Silicon Graphics

Run Time Type Identification for C++.....313  
*Bjarne Stroustrup, AT&T Bell Laboratories, Dmitry Lenkov, Hewlett-Packard California Language Laboratory*

Panel Discussion with Mark Linton and others

Run Time Type Information and Class Design.....341  
*Doug Lea, SUNY Oswego and Syracuse University*

# Smart Pointers: They're Smart, but They're Not Pointers

Daniel R. Edelson\*

edelson@sor.inria.fr

INRIA Project SOR, Rocquencourt BP 105, 78153 Le Chesnay Cédex, FRANCE

## Abstract

There are numerous times when a C++ user could benefit from a pointer variant that has more functionality than is provided by the basic, language-defined pointer. For example, type-accurate garbage collection, reference counting, or transparent references to distributed or persistent objects, might be implemented with classes that provide pointer functionality. The C++ language directly supports one kind of pointer substitute, the smart pointer, in the form of overloadable indirection operators: `->` and `*`.

In this paper we evaluate how *seamlessly* smart pointers can replace raw pointers. The ideal is for client code not to care whether it is using raw pointers or smart pointers. For example, if a `typedef` selects whether raw or smart pointers are used throughout the program, changing the value of the `typedef` should not introduce syntax errors.

Unfortunately, C++ does not support pointer substitutes well enough to permit seamless integration. This paper presents the desired behavior of smart pointers in terms of the semantics of raw pointers that the smart pointers try to emulate. Then, we describe several ways of implementing smart pointers. For each, we show cases in which the smart-pointers fail to behave like raw pointers. From among the choices, we explain which is the best for emulating the standard pointer conversions.

*Accessors* are similar to smart pointers, but have certain advantages. This paper discusses the differences between accessors and smart pointers, and shows why our conclusions about type conversion behavior also apply to accessors. Whether a programmer prefers smart pointers or accessors, this paper shows the limitations and recommends an implementation.

## 1 Introduction

The ability to substitute user-defined code for pointers is a very powerful programming mechanism. It facilitates using C++ in domains for which the language is not specialized. For example, smart pointers [Str87] or variations thereof can be used to support distributed systems [SDP92, SMC92], persistent object systems [MIKC92, SGH<sup>+</sup>89, Str91, pg. 244], to provide reference counting (e.g. the *ObjectStars* of [MIKC92] or the *counted pointers* idiom of [Cop92]) or garbage collection [Ken91, Ede92].

In this paradigm, a smart pointer encapsulates some kind of raw pointer or complex handle. The smart pointer overloads the indirection operators in order to be usable with normal pointer syntax. For example, code that accesses both transient and persistent objects can be written to perform its manipulations through smart pointers. These pointers would be

---

\*Author's other affiliation: Computer and Information Science, University of California, Santa Cruz CA 95064, USA, daniel@csc.ucsc.edu

This work has been supported in part by Esprit project 5279 *Harness*.

able to refer to either normal transient objects, or to objects that reside in persistent storage. When an object in persistent storage is referenced through the smart pointer, a copy is loaded into memory. The smart pointers should even be able to enforce a consistency protocol if the object is replicated or loaded into shared memory.

In analyzing how effective a pointer substitute is, we consider two criteria: (1) how run-time efficient it is, and (2), how it impacts the code in terms of programming style. Smart pointers with a lot of functionality could be quite inefficient; it is also possible to write very lightweight smart pointers. We do not concentrate on run-time efficiency because that is entirely determined by the specific implementation. Rather, we focus on the second issue: how the use of smart pointers impacts the client code.

This paper shows how the behavior of smart pointers diverges from that of raw pointers in certain common C++ constructs. Given this, we conclude that the C++ programming language does not support seamless smart pointers: smart pointers cannot transparently replace raw pointers in all ways except declaration syntax. We show that this conclusion also applies to *accessors* [Ken91].

The organization of this paper is as follows: Section 2 very briefly summarizes the behavior of raw pointers that smart pointers try to emulate, particularly in terms of the standard type conversions. Then, Sect. 3 presents several ways of implementing smart pointers, and for each, shows limitations and problems with it. Section 4 shows why these results apply equally to accessors, after which the last section concludes the paper.

## 2 Raw Pointer Behavior

In order to evaluate the effectiveness of a pointer substitute, it is necessary to have a baseline for comparison. That baseline is, of course, the *raw pointer*.<sup>1</sup> The semantics of raw pointers are too complex to list exhaustively. The most important aspect of their behavior for this discussion is how they undergo implicit type conversions. The problem is to design user-defined pointers that will behave nearly the same as raw pointers, in terms of implicit type conversions, in all interesting cases.

Table 1 summarizes the conversions that take place on function arguments and in expressions such as assignment. All of these type conversions may be performed *implicitly* by the compiler. We are not interested in *explicit* type coercions.

## 3 Smart Pointers

Smart pointers are class objects that behave like raw pointers [Str87, Str91]. The smart pointers overload the indirection operators (\* and ->) in order to be usable with normal pointer syntax. They have constructors that permit them to be initialized with raw pointers such as new returns. Smart pointers may supply a conversion to void\* in order to be usable directly used in control statements, e.g. if (ptr) and while (ptr). The conversion to void\* may also be seen as undesirable [Gau92], in which case all testing is explicit using overloaded comparison operators. Smart pointers may optionally supply a conversion to the corresponding raw pointer types.

Our goal in manipulating smart pointers is to have all the functionality of regular pointers *and then some*. For example, the 'and then some' might be:

<sup>1</sup>We use *raw pointer* to mean the pointer type that is directly supported by the compiler.

Table 1: Summary of implicit type conversions

The conversion classes are listed in order of precedence. The conversions within a group are of approximately the same precedence.

**Class 0: Trivial Conversions**

	From	To	Notes
1.	T	T&	<i>object</i> $\Rightarrow$ <i>reference</i>
2.	T&	T	<i>reference</i> $\Rightarrow$ <i>object</i>
3.	T[]	T*	<i>array</i> $\Rightarrow$ <i>pointer</i>
4.	T(args)	T(*) (args)	<i>function</i> $\Rightarrow$ <i>pointer</i>
5.	T	const T	<i>type</i> $\Rightarrow$ <i>const type</i>
6.	T	volatile T	<i>type</i> $\Rightarrow$ <i>volatile type</i>
7.	T*	const T*	<i>pointer</i> $\Rightarrow$ <i>pointer to const</i>
8.	T*	volatile T*	<i>pointer</i> $\Rightarrow$ <i>pointer to volatile</i>

**Class 1: Standard Conversions**

	From	To	Notes
9.	0	T*	<i>the NULL pointer conversion</i>
10.	Derived*	Base*	<i>if base is accessible and derived isn't const or volatile</i>
11.	Derived&	Base&	<i>if base is accessible and derived isn't const or volatile</i>
12.	T[]	T*	<i>array</i> $\Rightarrow$ <i>pointer to first element</i>
13.	T(args)	T(*) (args)	<i>except following &amp; or before ()</i>
14.	T*	void*	<i>provided T is not const or volatile</i>
15.	T(*) (args)	void*	<i>provided sufficient bits are available</i> [ANS91, §4.6, line 6]

**Class 2: User-defined Conversions**

- |     |  |
|-----|--|
| 16. | <i>conversion by constructor</i>         |
| 17. | <i>conversion by conversion operator</i> |

- tracing garbage collection [Ede92],
- reference counting [Ken91, Mae92, MIKC92, Cop92],
- convenient access to persistent objects [SGH<sup>+</sup>89, Str91, HM90, SGM89, MIKC92],
- uniform access to distributed objects [SDP92, Gro92, SMC92],
- instrumenting (measuring) the code,
- or others.

To accomplish this, the smart pointers should look and feel, to the greatest extent possible, like raw pointers. Achieving the ideal, i.e. making the smart pointer semantics a superset of raw pointer semantics, is impossible (as we will show). The next best thing is to see how close the code can come to making the smart pointers perfect substitutes for raw pointers in all ways except declaration syntax.

Raw pointers support numerous conversions, for example, conversion of T\* to void\*, of T\* to const T\*, and of derived\* to base\*. There are two ways to define smart pointers that can allow them to emulate these conversions:



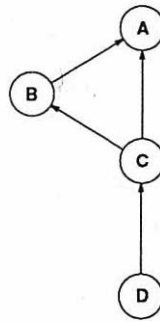


Figure 1: A sample class hierarchy

This hierarchy is rooted, but it need not be.

Sections 3.1.1 through 3.1.3 require that class D be in the hierarchy. However, figures later in the paper will only include classes A, B and C.

- 
1. the smart pointer classes can use user-defined type conversions to emulate the standard conversions, or,
  2. the smart pointer classes can be related in an inheritance hierarchy.

In this paper we will consider both these possibilities, sub-possibilities of each, and combinations thereof.

### 3.1 Supporting Class Hierarchies

Pointers in a class hierarchy undergo a very important set of conversions. In particular a derived class pointer can be implicitly converted to a base class pointer for an accessible base class. This conversion, along with virtual functions, is how C++ supports polymorphism. To be general, the smart pointer classes must emulate this type conversion.

#### 3.1.1 User-Defined Conversions

First we consider the case where the smart pointer classes do not have any subclass relations, even though the referenced classes may derive from each other. In this case, the standard conversions of raw pointers must be emulated with user-defined conversions. In particular, we are concerned with line 10 in Table 1: the derived class pointer to base class pointer conversion.

Let us assume that the class hierarchy of user objects is as shown in Fig. 1. There are four client classes: A, B, C, and D. Since there are four client classes we also require four smart pointer classes. We call the pointer classes Pa, Pb, Pc, and Pd. With standard conversions and raw pointers, the following implicit conversions are available:

$B^* \Rightarrow A^*$	$C^* \Rightarrow A^*$
$D^* \Rightarrow A^*$	$D^* \Rightarrow B^*$
$C^* \Rightarrow B^*$	$D^* \Rightarrow C^*$

The goal is to implement these same conversions among the smart pointer classes. Using user-defined conversions, there are two possibilities:

1. every smart pointer class provides a user-defined conversion to the smart pointer types that correspond to its referent type's direct bases, or,
2. every smart pointer class provides a user-defined conversion corresponding to every base class, whether direct or indirect.

### 3.1.2 Conversion to Direct Bases

Suppose every smart pointer class supplies a user-defined conversion to the smart pointer classes for direct base classes of the referent type. In our current example, this would provide the following user-defined conversions:

$P_b \Rightarrow P_a$	$P_c \Rightarrow P_b$
$P_c \Rightarrow P_a$	$P_d \Rightarrow P_c$

Under this scheme, there is no implicit conversion from  $P_d$  to  $P_a$ . This is because user-defined conversions can't be implicitly chained together. By contrast, with raw pointers the corresponding conversion is available. The failure to support conversion to an indirect base pointer is a substantial shortcoming of this implementation.

### 3.1.3 Conversion to All Bases

Instead of supplying user-defined conversions only to direct bases, we can instead provide conversions to all bases, direct and indirect. This scheme requires the following user-defined conversions:

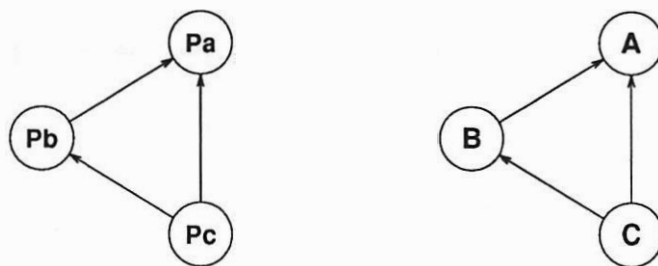
$P_b \Rightarrow P_a$	$P_c \Rightarrow P_b$
$P_c \Rightarrow P_a$	$P_d \Rightarrow P_a$
$P_d \Rightarrow P_b$	$P_d \Rightarrow P_c$

This supplies the conversion from  $P_d$  to  $P_a$  that was missing from the previous implementation. However, consider the following code:

```
void f(Pa);
void f(Pc);

int main(void) {
    Pd pd = new D;
    f(pd);
    return 0;
}
```

The call to  $f()$  is ambiguous. There are conversions to match both of the overloaded functions and there is no way to choose between them. In contrast, the equivalent code with raw pointers is unambiguous because, with raw pointers, conversion to a direct base is preferred over conversion to an indirect base, thus,  $f(C^*)$  would be called. This problem is less severe than the problem identified in §3.1.2; this is a more viable implementation.



A, B, C:	User classes
Pa, Pb, Pc:	Smart pointer classes for A, B, and C
$A \longrightarrow B$	Public virtual derivation of B from A

Figure 2: A pointer hierarchy for an object hierarchy

### 3.2 Inheritance Hierarchy

In the previous section, we discussed emulating the standard pointer conversions with user-defined conversions. It is also possible to emulate them using the standard reference conversions [Ken91]. We arrange the smart pointer classes in a class hierarchy that parallels the object hierarchy. Figure 2 illustrates this.

Since class Pc derives from Pb, any instance of Pc can be converted to an instance of Pb through the standard `Derived& to Base&` conversion. This reference conversion from Pc to Pb can thus be used to emulate the corresponding standard pointer conversion from `C*` to `B*`. The reference conversion has the same precedence as the pointer conversion, and also favors conversion to a direct base class over conversion to an indirect base class.

This scheme emulates the usual base class/derived class pointer conversions as follows. Assume that an instance of Pc (as shown in Fig. 2) must be converted to an instance of Pb, perhaps to initialize a temporary or to match a function parameter. Since class Pc is derived from class Pb, an instance of Pc contains an instance of Pb as a subobject. The standard conversion (Table 1, line 11) converts the Pc object to a Pb object by using the Pb subobject in place of the complete object. No user-defined code needs to be or may be provided to perform this conversion. The conversion simply changes the 'logical' address of the object from the beginning of the object to the beginning of the Pb subobject.

In an inheritance hierarchy of smart pointers, there is a choice to be made: What class defines the pointer instance data? Every class could potentially declare a pointer data member. Alternatively, either the root of the hierarchy or some other class can provide the data.

#### 3.2.1 Replicated Data

It is plausible for every smart pointer class in the smart pointer class hierarchy to define a new data member. Any derived class smart pointer then contains one pointer member added by the derived class, plus one pointer member for every direct or indirect base class. For example, suppose that Pb is a subclass of Pa, then a Pb contains a `B*` and the Pa subobject contains an `A*`.

A derived class smart pointer contains a subobject for each of its base classes; converting a derived class smart pointer to a base type uses the corresponding base class subobject in place of the complete object. After a conversion, the overloaded operators (such as indirection) use the base class pointer member rather than the derived class pointer member. To correctly emulate raw pointers, these base class pointers must all point into the same object as the main derived pointer, which is also called the *most derived* pointer. Therefore, assigning to a smart pointer under this implementation must update all of the component pointers. Failure to do this results in a derived class smart pointer that cannot be correctly converted to a base class smart pointer.

This implementation does not require any explicit type conversions, and emulates the standard pointer conversions well: conversion of a smart pointer to a base class smart pointer favors conversion to a direct base over conversion to an indirect base; this eliminates the problem discussed in §3.1.3 in which a choice between converting to a direct base or an indirect base is ambiguous. It also works correctly in the presence of multiple inheritance. However, it's inefficient because updating a derived class smart pointer requires an operation per base class. In addition, this scheme permits an incorrect type conversion. The alternative described in the following subsection suffers from the same error, so we defer the discussion until §3.2.3.

### 3.2.2 Nonreplicated Data

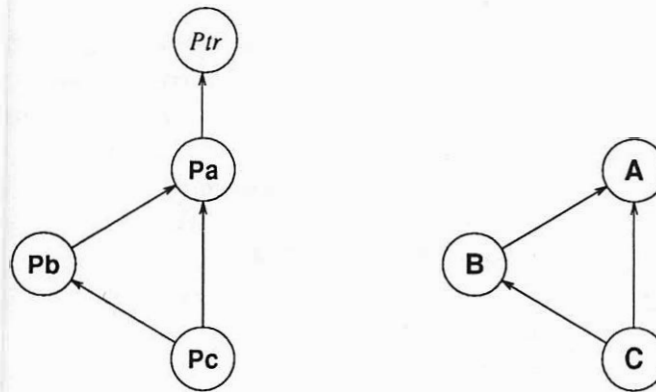
To improve the efficiency of the previous organization, we make every smart pointer contain exactly one pointer as its instance data. This is done by defining an abstract virtual base class that supplies the pointer datum; call this class *Ptr*. (Smart pointer classes that are indirectly derived from *Ptr* need not also be directly derived from it.) This way, each smart pointer class contains only one instance pointer. It also contains invisible pointers that implement the virtual derivation, but these pointers don't get modified during assignment. Figure 3 demonstrates this organization.

Since they use a virtual base class, under most C++ implementations, these smart pointers will have size larger than one word. Nonetheless, in contrast with the previous solution, assigning to one of these smart pointers only requires one indirect memory reference.

This organization supports conversion to a direct or indirect base class, and conversion to a direct base is preferred. Conversion to a base pointer is also preferred over conversion to `void*`. However, these smart pointers do not work with multiple inheritance.

Under multiple inheritance (and some implementations of single inheritance) a base class subobject may have a nonzero offset within a derived class object. With raw pointers, converting a derived pointer to a base pointer for such a base class adds the correct offset to the value of the pointer; this redirects the pointer from the beginning of the main object to the beginning of the base class subobject. For example, in Fig. 3, an object of class C contains a subobject of class B whose offset is probably nonzero. Converting a `C*` to a `B*` redirects the pointer from the beginning of the C object to the beginning of the B subobject by adding a positive offset to the pointer.

The corresponding conversion is performed on these smart pointers using the standard `derived& to base&` conversion shown on Line 11 of Table 1. The conversion causes the base smart pointer subobject to be used in place of the derived smart pointer object. This does not add the requisite offset to the value of the pointer. Instead, it simply reinterprets the same pointer value as a pointer of the base class type. Thus, these smart pointers cannot be converted to base class smart pointers for subobjects with nonzero offsets.



A, B, C:        User classes  
 Pa, Pb, Pc:    Smart pointer classes for A, B, and C  
 A  $\longrightarrow$  B    Public virtual derivation of B from A  
 Ptr:            Base class to supply the pointer datum

Figure 3: A smart pointer hierarchy with an abstract base to supply the data

Expressed differently, the problem is that the operation `derived::operator base&()` cannot be overloaded. This is a built-in standard conversion that causes the compiler to substitute the base subobject in place of the derived object. If this operator could be overloaded such that it altered the value of the pointer, then the error could be avoided. Note, the current language definition does not explicitly forbid overloading this operator, nor does it explicitly permit it [ANS91], however, it seems inevitable that overloading this operator will eventually be prohibited.

### 3.2.3 Another Error with Pointer Hierarchies

There is one other error that the schemes presented in the last two subsections both share: they both permit an incorrect, implicit type conversion.

Every C++ programmer is familiar with the conversion from `Derived*` to `Base*`. However, the conversion from `Derived**` to `Base**` is prohibited because it introduces a gaping hole in the otherwise (mostly) safe type system. Specifically, given two objects whose classes are different but have a common base, this conversion allows you to incorrectly compare or assign pointers to these objects [Sal92]. Figure 4 provides an example of how this conversion allows assignment between two incompatible pointer types.

With a class hierarchy of smart pointers, this conversion is not just between `Derived**` and `Base**`; it is also between `Derived*` and `Base*` because the smart pointer classes are related through inheritance. The compiler permits the conversion because it uses the standard base class pointer conversion listed on Line 10 of Table 1. Figure 5 shows the same incorrect code using smart pointers. The difference is that the code using smart pointers compiles without error and crashes at runtime.

To show that this error also occurs with accessors, the code of Figure 6, written using OATH accessors and library classes, encounters this bug and dies with a segmentation

```

class BASE { ... };
class DER1 : public BASE { ... };
class DER2 : public BASE { ... };

void f(BASE** p1, BASE** p2) { *p1 = *p2; }

int main(void)
{
    DER1 * d1 = new DER1;
    DER2 * d2 = new DER2;

    f(&d1,&d2); // Illegal, but what if?
    return 0;
}

```

Figure 4: Why a `derived**` may not be converted to a `base**`

If a `derived**` could be converted to a `base**`, then this code would assign a `DER1*` to a `DER2*`. However, there is no relationship between classes `DER1` and `DER2` that would justify such an assignment.

---

violation. This error exists because the pointer hierarchy provides the incorrect conversion of `Derived**` to `Base**`. (For those readers not acquainted with OATH accessors, there is a discussion of the differences between them and smart pointers in §4.)

### 3.2.4 Class Hierarchies Summary

We have presented four ways of organizing smart pointers to support class hierarchies. These ways include two that depend on user-defined conversions and two that use a parallel class hierarchy.

With user-defined conversions, it's best to supply conversions to both direct and indirect base classes. Given that, the problem is that the compiler can't choose between converting to a direct base and converting to an indirect base, nor between converting to a base class and converting to `void*`. Consequently, certain overloaded function invocations are ambiguous, whereas they are legal using raw pointers.

The alternative to user-defined conversions is to use a parallel class hierarchy of smart pointers. This uses standard reference conversions to convert a derived class smart pointer to a base class smart pointer. It is inefficient to replicate the pointer data in each class, so an abstract base class is used to supply a `void*` instance datum. However, this scheme does not support multiple inheritance, and it permits an incorrect pointer conversion.

Of the possibilities discussed, we suggest using user-defined type conversions to direct and indirect base classes. The programmer may need to disambiguate some overloaded function calls that would be legal using raw pointers.

```

void f(PtrBASE* p1, PtrBASE* p2) { *p1 = *p2; }

int main(void)
{
    PtrDER1 d1 = new DER1;
    PtrDER2 d2 = new DER2;

    // Legal and wrong with a pointer hierarchy
    f(&d1,&d2);
    return 0;
}

```

Figure 5: The invalid conversion with smart pointers

Since the smart pointer classes are related through inheritance, the compiler permits the type conversion, even though this results in an assignment between incompatible types.

### 3.3 Supporting const

Supporting the base class conversions is one problem. Supporting the conversion of `T*` to `const T*` is equally or more important because of the major role that `const` plays in documenting and structuring C++ programs.

Using raw pointers, there are two ways to modify a pointer declaration using `const`:

1. `const T*`                      *The referent is const.*
2. `T* const`                      *The pointer is const.*

These uses of `const` are not mutually exclusive, thus, `const T* const` is the type of a pointer for which both the referent and the value are `const`.

With smart pointers, on the other hand, `const` only can be used one way: `const PtrT ptr`. This does not declare a smart pointer to a `const` object. Rather, this declares a smart pointer whose value may not change. The reader may argue that this discussion does not apply given templates because with templates we can declare both `Ptr<T>` and `Ptr<const T>`. However, these are two distinct types. This is the same as hand coding two classes: `Ptr_T` and `Ptr_const_T`. Being defined from the same template does not give the two classes any special relationship. In particular, there is no implicit type conversion from `Ptr<T>` to `Ptr<const T>`.

For this reason, one class of smart pointer cannot reference both `const` and mutable objects; instead, we need two smart-pointer classes. Let `PtrT` be the smart pointer class that replaces pointers of type `T*`, and let `CPtrT` be the smart pointer class that replaces pointers of type `const T*`. An overloaded indirection operator of `CPtrT` returns a `const` object; this allows the compiler to complain about attempts to modify an object through a `CPtrT`. For these smart pointers to resemble raw pointers, there must be a conversion from `PtrT` to `CPtrT`.

The conversion from `PtrT` to `CPtrT` can be implemented two ways: either there can be a user-defined conversion between them, or `PtrT` can be a derived class of `CPtrT`. The use



```

#include <iostream.h>
#include "oath/minString.h"

void f(objA & a, objA & b) { a = b; }

int main(void)
{
    characterA ch = characterA::make('A');
    stringA str = minStringA::make();

    str << "hello\n";
    cout << str;
    f(str,ch);      // incompatible assignment
    cout << str;    // This causes a core dump.
    return 0;
}

```

Figure 6: How to misuse the conversion that smart pointer hierarchies permit

This example uses OATH accessors, which are discussed in §4.

---

of the user-defined conversion is self-explanatory. If the one is a derived class of the other, then the standard reference conversion can be used in place of the normal standard pointer conversion, as we have described previously.

Assume that the conversion between the two smart pointer classes is user-defined. Here are two classes of code that are affected:

1. The following works fine with raw pointers, but when the conversion from `PtrT` to `CPtrT` is user-defined, the code is illegal because it requires two user-defined conversions.

```

struct S {
    S(CPtrT);
};

S func(PtrT p) { return p; }

```

2. If a function is overloaded on types `void*` and `CPtrT`, it cannot be invoked with a `PtrT` because the call would be ambiguous. With raw pointers, the call would favor conversion to `const T*` over conversion to `void*`.

A better way to implement the `const` pointer conversion is to make the class `PtrT` a derived class of `CPtrT` through public non-virtual derivation. They can share the same pointer data member so that instances of each class occupy only one word of storage. Figure 7 presents the basic structure of this organization. This uses a standard reference conversion to emulate the standard pointer conversion. The difference will be unnoticeable for most programs, except for the declaration syntax.

```

// smart pointer class to replace 'const T *'
class CPtrT {
protected:
    union {
        T * ptr;
        const T * cptr;
    } value;
public:
    ...
};

// smart pointer class to replace 'T *'
class PtrT : public CPtrT {
public:
    ...
};

```

Figure 7: A smart pointer hierarchy for const

### 3.4 Overall

We have identified 7 properties that a smart pointer organization should provide. They are (with keywords for future reference):

- dir** implicit conversion to a direct base pointer;
- indir** implicit conversion to an indirect base pointer;
- prefer** a preference for converting to a direct base over an indirect base;
- mult** support for multiple inheritance;
- safe** no conversion from `derived**` to `base**`;
- const** the ability to reference normal and `const` objects, with compiler enforcement of the `const` attribute, and a conversion from non-`const` to `const`;
- fast** the organization should be intrinsically efficient.

In general, any type conversion among the smart pointers should be the same precedence as the conversion to which it corresponds among raw pointers. For example, the conversion from `derived*` to `base*` is Class 1 (a *standard* conversion), as shown in Table 1. Therefore, it would be best for the corresponding conversion among smart pointers also to be Class 1. If this is done, the smart pointers closely resemble raw pointers in terms of overloaded function resolution and implicit conversions. Table 2 shows how well each organization that we've presented satisfies these goals.

As shown in Table 2, a class hierarchy of smart pointers emulates the derived class/base class conversion and the `const` pointer conversion well. However, it only supports inheritance when all subobjects have offset zero, and thus it fails to support multiple inheritance. In addition, it introduces the erroneous `derived**` to `base**` conversion. Therefore, a class hierarchy of smart pointers is good for implementing the `const` conversion, but not for implementing the base class conversions.

Table 2: Strengths and weaknesses of these methods

Method	dir	indir	prefer	mult	safe	const	fast
userdef direct (§3.1.2)		—	—	+	+		+
userdef all (§3.1.3)			—	+	+		+
hier replicated (§3.2.1)	+	+	+	+	—	+	—
hier abstract (§3.2.2)	+	+	+	—	—	+	+
recommended hybrid (§3.4)			—	+	+	+	+
OATH accessors (§4)	+	+	+	—	—	—	+

+ good behavior

[ ] a user-defined conversion replaces a standard one

— incorrect behavior

By contrast, user-defined conversions are less desirable in all cases because they replace a standard or trivial conversion with a user-defined conversion; this difference is noticeable in terms of overloaded function resolution and chaining of type conversions. In spite of that disadvantage, however, user-defined conversions allow the smart pointers to support the base class/derived class conversion, even under multiple inheritance, and don't permit the erroneous conversion.

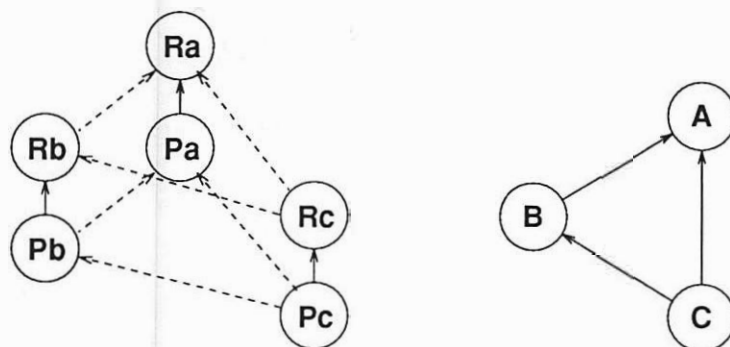
These two observations lead to our recommended overall organization. We suggest using user-defined conversions to emulate the base class/derived class conversions because this is safe and correct. Simultaneously, the smart pointers should use a smart pointer inheritance hierarchy to emulate the `const` conversions.

A diagram of this organization is shown in Fig. 8. This shows an application class hierarchy and the corresponding smart pointer classes, including both the smart pointer classes for regular objects, and those for `const` pointers. For each of the application's classes there are two smart pointer classes, one that references mutable objects and one that references `const` objects. The smart pointer class that references mutable objects is a derived class of the one that references `const` objects. This supplies a standard conversion from *pointer to mutable* to *pointer to const*. In addition, the smart pointer classes for distinct application classes are related through user-defined type conversions. If class B is a derived class of A, then Pb provides a user-defined type conversion to Pa, and CPb provides a user-defined type conversion to CPa. (CPb is the smart pointer class for `const` Bs.)

The use of user-defined conversions between distinct types `PtrX` and `PtrY` supports multiple inheritance and avoids the erroneous conversion. The classes `PtrX` and `CPtrX` are related by inheritance because it gives better behavior without allowing false conversions; the compiler can correctly enforce the `const` attribute of a referent of `CPtrX`.

### 3.4.1 A Unrooted Hierarchy

While we have only discussed using the smart pointers in a class hierarchy with a unique root, this does not make any difference in the implementation that has been suggested. Any type conversion that is legal among raw pointers can be implemented by the smart



A, B, C:        User classes  
 Pa, Pb, Pc:    Smart pointer classes for A\*, B\*, and C\*  
 Ra, Rb, Rc:    Smart pointer classes for const A\*, etc.  
 —————>    Public derivation  
 - - - - ->      User-defined type conversion

Figure 8: The final smart pointer organization for the indicated object classes.

pointers by encapsulating the raw pointer conversion within a user-defined type conversion. Of course, as we have mentioned, whenever a user-defined conversion replaces a built-in conversion, some cases of overloading and chaining of conversions do not behave as desired.

### 3.5 Other Weaknesses

#### 3.5.1 Pointers to volatile Objects

This paper has discussed `const`, but not `volatile`. Pointers to `volatile` objects must be supported in exactly the same way as pointers to `const` objects. In particular, for a single application class, distinct smart pointer classes are required to reference:

1. normal objects
2. `const` objects
3. `volatile` objects
4. `const volatile` objects

This plethora of classes adds a certain amount of notational complexity to the program.

#### 3.5.2 Conversion Precedence

The proposed organization appears to be the best of the ones that have been considered because it is both safe and efficient. However, it emulates the standard `derived*` to `base*` conversions with user-defined type conversions. User-defined type conversions have lower precedence than the standard conversions. Therefore, there are many situations, primarily

Table 3: Some ways in which our smart pointers don't behave like raw pointers.

Case	Raw Pointers	Smart Pointers
Convert either to <i>pointer to direct base</i> or to <i>pointer to indirect base</i>	Convert to direct base	Ambiguous
Convert either to <i>pointer to base</i> or to <i>void*</i>	Convert to base	Ambiguous
Chain conversion to <i>pointer to base</i> with another user-defined type conversion	Legal	Illegal

involving function overloading, in which these smart pointers do not behave the same as the corresponding raw pointers. Table 3 lists some of the cases in which these smart pointers behave differently than raw pointers.

### 3.5.3 Pointer Leakage

It is essentially impossible to prevent smart pointers from leaking raw pointers to the application (e.g. this pointers). In some cases, it is desirable to prevent this. For example, if smart pointers are used to implement copying garbage collection, then after a garbage collection, all dynamically allocated objects have been moved and any raw pointer no longer has the correct value.

As another example, [Ken91] discusses why the problem of raw pointer leakage makes smart pointers unsafe for reference counting. The basic idea is that the application can obtain reference counted pointer as a temporary expression, perhaps as the return value from a function. The application may then dereference the reference counted pointer by invoking the overloaded operator `->`, which returns a raw pointer, which will in turn be dereferenced. Once the raw pointer is returned from the overloaded operator `->`, the reference counted pointer has served its purpose and may be destroyed. However, destroying the reference counted pointer decrements the object's reference count and may cause the object to be deallocated. If the object is deallocated, then the raw pointer, which is about to be dereferenced, is a dangling reference.

In other cases, it is not critical that the application be prevented from obtaining raw pointers. For example, mark-and-sweep garbage collectors can normally tolerate the existence of raw pointers, provided the raw pointers point at objects that are *also* referenced by smart pointers [Ede92].

Smart pointers leak raw pointers because of the definition in C++ of the overloaded indirect member access operator, `->`. When the compiler sees an expression of the form `X->Y`, where `X` is an expression of class type, the compiler evaluates `X.operator->()`. The language definition requires that this operator return a raw pointer.<sup>2</sup> This is a potential problem because if the smart pointer was a temporary object, the compiler may destroy it as soon as the raw pointer is obtained. However, as shown for the case of reference counting, for example, destroying the smart pointer may cause the raw pointer to become a dangling reference. This is the main problem that accessors solve.

<sup>2</sup>These operators may be chained together, but must eventually return a raw pointer.

```

// A sample application class.
class Thing {
friend class ThingA;
private:
    int value;
    Thing(int initial) : value(initial) { }
    void set(int val) { value = val; }
    int get()          { return value; }
    ...
};

// A class for accessing Things.
class ThingA {
private:
    Thing * ptr;
public:
    ThingA() : ptr(0) { }
    void make(int i) { ptr = new Thing(i); }

    void set(int i) { ptr->set(i); }
    int get()       { return ptr->get(); }
    ...
};

```

Figure 9: An object class and an *accessor*-type reference class

The accessor class contains a raw pointer as its instance datum. All of the client class' member functions are duplicated in the accessor class and accessed with `'.'`. Therefore, the accessor class does not need to overload the indirection operators.

---

## 4 Accessors

Kennedy describes accessors in OATH [Ken91] as an alternative to smart pointers. The central difference between accessors and smart pointers is that accessors don't overload the indirection operators; instead, like stubs [DMS92], they duplicate all the public member functions of the referent object and forward those calls through a pointer to the object. Accessors are somewhere in between smart pointers and *smart references*, because they implement pointer semantics, but use `'.'` rather than `'->'` to access the underlying object. Figure 9 gives the general idea behind how accessors work. This figure does not attempt to reproduce all the functionality described in [Ken91], instead, it just shows the relation between the application class and the accessor class.

Accessors are clearly superior to smart pointers because they prevent raw pointer leakage. However, they are difficult to declare because every member function of the application class must also be declared in the accessor class. Macros can abbreviate this, but the code

looks significantly different from standard C++ class definitions and complex macros can hinder debugging.

The accessors in OATH are organized into a class hierarchy that parallels the data object hierarchy. The reference conversions are used to convert one accessor class into a different one. The class hierarchy is rooted in the class `oathCoreA`; it is this class that supplies the pointer data member. This organization was discussed in Sect. 3.2.2. (Indeed, it was OATH that led us to consider this organization.)

The OATH class hierarchy uses only single inheritance; the class hierarchy, therefore, forms a tree. If it used multiple inheritance, then its implementation would suffer from the incorrect offset problem described in 3.2.2. In particular, for a pointer conversion that changes the value of the pointer, the corresponding reference conversion is incorrect because it changes the type of the accessor without changing the value of the pointer. Even using only single inheritance, this scheme permits the incorrect type conversion of `derived**` to `base**` that we discuss in 3.2.3 (see Fig. 6). Finally, the hierarchy of OATH uses a single accessor class per object class; therefore, it is unable to represent pointers to `const` objects (§3.3).

Accessors suffer from the same problems, with respect to type conversions, as smart pointers. However, the accessor model is safer than the smart pointer model. By not overloading `->`, accessors avoid leaking raw pointers in a way that may result in dangling references if the compiler is aggressive in destroying temporary objects.

## 5 Conclusion

Pointer substitutes, whether smart pointers or accessors, are a powerful programming paradigm. C++ supports them, but not to the extent of allowing them to integrate seamlessly into a program. There are two main limitations: (1) supporting pointers to `const` objects, and (2) supporting the standard pointer conversions.

We have presented several possible implementations, and discussed how they address these two limitations. Supporting pointers to `const` objects requires two smart pointer classes per object class. The two smart pointer classes should be defined such that the class for the pointer to mutable is derived from the class for pointer to `const`. Supporting class hierarchies is more difficult. The best way appears to be to use user-defined type conversions between the pointer classes. The behavior under this organization diverges from that of raw pointers in some circumstances that involve function overloading or chaining user-defined conversions. However, this should present only a slight inconvenience, not a fatal handicap.

Changes to C++ could allow it to support smart pointers better. Some possible changes include allowing some user-defined conversions to chain, or permitting user-defined code to implement the `derived::operator base&()` conversion. However, smart pointers are useful enough that it's important to identify how best to implement them, given the current language definition. That's what this paper has done: We've shown how to make smart pointers closely emulate the standard pointer conversions for `const` and class hierarchies, while circumventing erroneous and incorrect type conversions.

## Acknowledgements

I would like to thank: Peter Dickman, David Plainfossé, Darrell Long and Marc Shapiro for commenting on various versions of the paper, the conference referees for



their relevant insightful comments, **Philippe Gautron** for some lively discussions and comments on the paper, and, **Marc Shapiro** (again) for supporting this work while I'm in the SOR group at INRIA, Rocquencourt.

## References

- [ANS91] ANSI X3J16/ISO WG21 working document X3J16/91-0115, May 1991. Draft ANSI/ISO standard for the C++ programming language.
- [Cop92] James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.
- [DMS92] Peter Dickman, Messac Makpangou, and Marc Shapiro. Contrasting fragmented objects with uniform transparent object references for distributed programming. In *SIGOPS 1992 European Workshop on Models and Paradigms for Distributed Systems Structuring*, 1992.
- [Ede92] Daniel R. Edelson. Precompiling C++ for garbage collection. In *International Workshop on Memory Management*, 1992. *To appear*.
- [Gau92] Philippe Gautron. Don't convert smart pointers to void\*, 1992. Private communication.
- [Gro92] Ed Grossman. Using smart pointers for transparent access to objects on disk or across a network. private communication, 1992.
- [HM90] Antony L. Hosking and J. Eliot B. Moss. Towards compile-time optimizations for persistence. In *4<sup>th</sup> Inter. Workshop on Persistent Object Systems*, pages 17–27, 1990.
- [Ken91] Brian Kennedy. The features of the object-oriented abstract type hierarchy (OATH). In *Usenix C++ Conference Proceedings* [Use91], pages 41–50.
- [Mae92] Roman E. Maeder. A provably correct reference count scheme for a symbolic computation system. In unpublished form, 1992.
- [MIKC92] Peter W. Madany, Nayeem Islam, Panos Kougouris, and Roy H. Campbell. Reification and reflection in C++: An operating systems perspective. Technical Report UIUCDCS-R-92-1736, Dept. of Computer Science, University of Illinois at Urbana-Champaign, March 1992.
- [Sal92] Hayssam Saleh. *Conception et réalisation d'un système pour la programmation d'applications objets concurrentes et réparties sur machines parallèles*. PhD thesis, Université Pierre et Marie Curie—Paris VI, 1992.
- [SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Symp. on Principles of Distributed Computing*, Vancouver (Canada), August 1992. ACM.
- [SGH<sup>+</sup>89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

- [SGM89] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In Stephen Cook, editor, *EC'OO'89, Proc. of the Third European Conf. on Object-Oriented Programming*, British Computer Society Workshop Series, pages 191-204, Nottingham (GB), July 1989. The British Computer Society, Cambridge University Society.
- [SMC92] Marc Shapiro, Julien Maisonneuve, and Pierre Collet. Implementing references as chains of links, 1992. *Submitted to IWOOS '92*.
- [Str87] Bjarne Stroustrup. The evolution of C++ 1985 to 1987. In *Usenix C++ Workshop Proceedings*, pages 1-22. Usenix Association, November 1987.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2<sup>nd</sup> edition, 1991.
- [Use91] Usenix Association. *Usenix C++ Conference Proceedings*, April 1991.



# Not A Language Extension

Martin D. Carroll  
carroll@mozart.att.com

AT&T Bell Laboratories  
184 Liberty Corner Road  
Warren, New Jersey 07059-0908

This is a paper about what might have been proposed as a C++ language extension.

People propose language extensions when they discover what they believe is an important class of problems which has no acceptable solution in the current language. Unfortunately, sometimes an acceptable solution can be surprisingly difficult to discover. This fact, together with the presence of direct support in *other* languages for the given class of problems, may inspire a programmer to think about language extensions before looking hard enough for a solution in the current language.

In this paper, we give an example of such a problem, that of *recursive iterators*. We show how a detailed analysis of the problem might lead a programmer to propose a language extension. We then present an easily reusable solution which requires no extension.

## 1 Introduction

The process of inventing an extension for a programming language can be represented by the following pseudo-code:

- (1) discover an important class C of problems
- (2) if there is no acceptable solution for C in the current language
- (3)     invent a language extension to solve C

Once we invent an extension, however, we are not done. Every language extension bears a cost in the complexity — time, space, and conceptual — it adds to the language. An extension is justified only if this complexity is worth being able to solve (or more easily solve) the targeted class of problems.<sup>1</sup> Much of the time spent arguing over proposed extensions is spent discussing this tradeoff.

Long arguments over this tradeoff can sometimes obscure step (2). A language extension that adds a non-trivial amount of complexity to a language is not worth adopting if there is already an acceptable way to solve the targeted class of problems in the current language.

This much is obvious. A mistake that can be easy to make is not spending enough time searching for an acceptable solution in the current language. But how much time is enough time? A day? A week? A year? If a solution takes a year to discover, doesn't that fact alone automatically disqualify it from being acceptable?

No. *An acceptable solution is not necessarily easy to discover.* Once a solution is discovered, it doesn't have to be *rediscovered* by every programmer who uses it. In fact, a solution that is hard to find may be viewed as "obvious" once it is discovered. (An example of such a solution can be found in [2].)

---

<sup>1</sup>The designers of Modula-3 took this to the limit when they decreed that a fundamental design goal of Modula-3 was to keep the reference manual under sixty pages.[1]

In this paper, we present an important class of problems, the problem of *recursive iterators*. We show how the temptation to propose a language extension for this class is great — especially because language support for this same class exists in other languages. Then, by returning to step (2), we discover a solution in the current language which, although not easy to find, is easy to reuse.

## 2 A Problem

Consider the problem of visiting the nodes of a graph in *depth-first search order*[3].

```
for each node n of a graph G in depth-first search order
do something with n
```

How do we implement that in C++? A common approach is to use an *iterator*. An iterator for this problem would return each of the nodes of G in depth-first search order.

```
Dfs_iterator i(G);
Node n;
while (i.next(n))
    // do something with n
```

The specific interface of this iterator does not concern us. What is important is that the user (1) declares an iterator object whose argument is the graph over which he wants to iterate; and (2) repeatedly calls a member function of the iterator (here named `next`) to get the next node in the ordering.

Let's try to implement `Dfs_iterator`. Implementing the constructor is easy.

```
class Dfs_iterator {
    Graph g;
public:
    Dfs_iterator(Graph G) : g(G) {}
    // ...
};
```

Here we just save a copy of the entire graph. In a real implementation, we'd use references, or pointers, or handles, or something else more efficient. For our purposes, it doesn't matter.

To implement the `next` function, we start with the classic algorithm for depth-first search [3].

```
dfs(Graph G) {
    clearmarks();
    visit(G, root(G));
}

visit(Graph G, Node n) {
    mark(n);
    for each successor m of n in G
        if (!marked(m))
            visit(G, m);
}
```

Unfortunately, this simple algorithm cannot be directly used to implement `next`. At each point in the computation when we enter `visit`, we will have the next node (`n`) to return to the client. Unfortunately, we will be deep in a recursion whose state must be remembered for the *next* call to `next`. If we simply return `n` to the user, we will lose this state.

We call this the problem of *recursive iterators*. Below we consider three possible solutions to this problem.

## 2.1 Solution 1: Remove the recursion

The first solution is to turn the recursive algorithm into a nonrecursive one that explicitly maintains the recursion stack. Although there is a relatively mechanical (that is, mechanical in theory but not in practice) procedure for doing this, it is tedious and error-prone, and produces unmaintainable code. Let's see if we can't find a better way.

## 2.2 Solution 2: Pre-iterate

The second solution is to perform the entire depth-first search upon the first call to `next`, store the resulting list of nodes in an internal list, and have successive calls to `next` return the next node on the list.

Such "pre-iteration" schemes waste space (the internal list) and time (if the client needs only an initial segment of the iteration). Further, for iterators that produce an infinite sequence of values (for example, a random number iterator), precomputing the entire list is not possible.

## 2.3 Solution 3: Import the iteration

Suppose the task the user needs to perform on each node is a simple function call.

```
// ...
while (i.next(n))
    do_something_with(n);
```

Then we could modify the depth-first search algorithm as follows:

```
dfs(Graph G, void (*f)(Node)) {
    clearmarks();
    visit(G, root(G), f);
}

visit(Graph G, Node n, void (*f)(Node)) {
    f(n);
    // ...
}
```

That is, every time we visit the next node, we perform the user's task on his behalf. If we now change the definition of the iterator to the following:

```
class Dfs_iterator {
public:
    Dfs_iterator(Graph G, void (*f)(Node)) { dfs(G, f); }
};
```

the user can simply say this:

```
Dfs_iterator(G, do_something_with);
```

Notice that *constructing* the iterator has the effect of performing the entire iteration. Such an iterator is called an *importing* iterator, because it “imports” the user’s function into the body of the iteration.

This may appear to be a reasonable solution to our problem. In fact, it may appear to be *superior* to our original interface, because the user doesn’t have to type as much. However, suppose the user’s original code had looked like this:

```
T t;
// ...
while (i.next(n)) {
    // do something with n
    // use t
}
```

Here the body of the iteration uses a variable declared outside the body. Now what function should the user pass to the constructor of the iterator?

The user must create a *new* function, and move the body of the loop to it.

```
void do_something_with(???) {
    // do something with n
    // use t
}
```

How do we pass the value of *t* to this function? There are only two choices: use a global variable, or use an argument.<sup>2</sup> Since we eschew global variables, especially for such a localized need as this, we use an argument.

```
void do_something_with(Node n, T t) {
    // do something with n
    // use t
}
```

Unfortunately, now this

```
Dfs_iterator(G, do_something_with);
```

doesn’t work. The type of *do\_something\_with* does not match the type of function expected by *Dfs\_iterator*’s constructor.

We might hope that turning *Dfs\_iterator* into a *template* might help.

```
template <class fcn_type>
class Dfs_iterator {
public:
    Dfs_iterator(Graph G, fcn_type f) { dfs(G, f); }
private:
    void dfs(Graph G, fcn_type f) { /* ... */ }
    void visit(Graph G, Node n, fcn_type f) { f(n); /* ... */ }
};
```

---

<sup>2</sup>There is actually a third choice: propose a language extension allowing nested functions [4].



This doesn't work, however, because the *call* of `f(n)` in `visit` still requires a function of type `void(*) (Node)` (or compatible type).

A solution that does work is to make `f` a *function object*. First we define a class that models the action of "applying a function to a node."

```
class Node_applicator {
public:
    virtual void operator()(Node) = 0;
};
```

Next we change `Dfs_iterator` to make it use a `Node_applicator`, rather than an actual function.

```
class Dfs_iterator {
public:
    Dfs_iterator(Graph G, const Node_applicator& f) { dfs(G, f); }
private:
    void dfs(Graph G, const Node_applicator& f) { /* ... */ }
    void visit(Graph G, Node n, const Node_applicator& f) { f(n); /* ... */ }
};
```

We must pass the `Node_applicator` by reference, because it is an abstract class. The call of `f(n)` in `visit` now works fine: it calls the virtual `Node_applicator::operator()`.

To use this iterator, the user must first derive his own `Node_applicator` class.

```
class Do_something_with : public Node_applicator {
public:
    Do_something_with(T t) : save_t(t) {}
    void operator()(Node n) { do_something_with(n, save_t); }
private:
    T save_t;
};
```

The constructor saves the value of `t` to be used by `do_something_with`; the `operator()` function makes the appropriate call, using the saved value.

Now the user can iterate over his graph as follows:

```
T t;
// ...
Dfs_iterator(G, Do_something_with(t));
```

This works. Indeed, the same approach can be made to work for *any* iterator. The question is, Was this solution acceptable? Although it was difficult for us to discover, now that we know about it, is it "obvious"?

No. For the iterator implementor, it's not too bad: just define the appropriate applicator class, and use it to call the user's function during the iteration.

For the iterator *user*, on the other hand, it's just *awful*. For even the simplest iterations, the user must create an actual function to hold the body of the iteration, and then derive a new class to call that function, remembering to pass in and save all the necessary variables in temporaries within that class. And that's not all: if the user ever needs to *change* the body of the iteration, he must potentially make changes to the *function*, the *class*, and the *call site* of the iterator.

There must be a better way.

### 3 Other Languages

When you can't figure out what to do, it often helps to look at other programming languages. Here is how a CLU [5] programmer would implement the depth-first search iterator:

```
// CLU code
(1) dfs = iter(G: Graph) yields(Node)
(2)     clearmarks()
(3)     for n: Node in visit(G, root(G)) do
(4)         yield(n)
(5)     end
(6) end dfs
(7) visit = iter(G: Graph, n: Node) yields(Node)
(8)     yield(n)
(9)     mark(n)
(10)    for each successor m of n in G do
(11)        if not marked(m)
(12)            visit(G, m)
(13)        end
(14)    end
(15) end visit
```

Except for some small differences, this is just the classic algorithm. Those differences (other than unimportant syntactic ones) are as follows:

(1) `dfs` and `visit` are each declared as an `iter`. This means they return a *sequence* of values rather than a single value.

(2) The type of the values returned by the iterators is declared by the `yields`. Here both `dfs` and `visit` return a sequence of `Nodes`.

(3) The values returned are those passed as arguments to `yield`. For example, if when it is executed `visit` above calls `yield(n1)`, `yield(n2)`, `yield(n3)`, ..., then the sequence of values returned is `n1, n2, n3, ...`.

(4) The user of an iterator retrieves the sequence of returned values with a for-loop construct. The loop on line (3) iterates over every value yielded by `visit`; each of these values is itself yielded by `dfs`. Similarly, the user of `dfs` would say the following:

```
// CLU code
for n: Node in dfs(G) do
    // do something with n
end
```

The way this is implemented in CLU is by running the iterator code and the user code as coroutines. Each iteration of the for-loop causes a thread context switch to the iterator; each `yield` in the iterator causes a context switch back to the loop, with the yielded value as the next value of the loop variable. When the iterator returns, the for-loop is exited.

This is a very nice solution, for both the iterator implementor and the user. It lets both write their code in a natural way. Of course, this ability does not come for free — the facility is built into the language — but the solution provided is so elegant, it seems worth it.

C++ has no such facility. The temptation to propose a language extension is great. We are ready to write ANSI. When we discover that languages other than CLU (for example, Icon [6] and the Pascal variant of [7]) also build in such features, we take pen in hand.

## 4 Solution 4: A Language Extension

J. Programmer  
January 17, 1992

Dear ANSI:

I have just discovered an important class of problems for which there is no acceptable solution in C++ as the language is currently defined. [*Here the problem of recursive iterators, and the CLU solution, are discussed.*]

Other languages let their programmers do this, why can't we? I therefore propose the following language extension to C++:

new keywords: `iter`, `yields`, `in`  
new standard library function: `yield`  
new syntactic construct: `for ([var] in [iterator]) [statement]`  
new meaning: just like CLU

Thank you for seeing your way clearly to adding this most important feature.

Sincerely,  
J. Programmer

## 5 More Problems<sup>3</sup>

ANSI Committee  
March 20, 1992

Dear J.:

What are you *crazy*?! C++ already has a bazillion features, and you want to add *another* one?! Go take your language extension and eat dirt, nano-brain!

Sincerely,  
ANSI Committee

## 6 Solution 5: A Template

We decide to take another approach. After spending a non-trivial amount of time thinking about this problem, we discover another solution. To make a long story short, we discover the following template:

```
template <class yields>
class Iter {
public:
    Iter();
    virtual ~Iter();
    int next(yields&);
protected:
```

---

<sup>3</sup>The committee wishes they could send the following reply. Instead, they'll have to be polite and send a "Thank you for sharing" letter.

```

        virtual void yielder() = 0;
        void yield(const yields&);
private:
    // ...
};

```

This template (as we shall see) lets us implement the depth-first search iterator as follows:<sup>4</sup>

```

class Dfs_iterator: public Iter<Node> {
public:
    Dfs_iterator(Graph G) : g(G) {}
private:
    Graph g;
    void yielder() { dfs(g); }
    void dfs(Graph g) {
        clearmarks();
        visit(g, root(g));
    }
    void visit(Graph g, Node n) {
        yield(n);
        mark(n);
        for ( each successor m of n in g )
            if (!marked(m))
                visit(g, m);
    }
};

```

This code, like the CLU code, is just the classic algorithm, with a few small differences. Those differences are as follows (compare this list with that on page 6):

(1) `Dfs_iterator` is derived from the abstract base class `Iter<Node>`. This means it is an iterator that will be returning its values via `yield` calls.

(2) The type of the values returned by the iterator is declared by the *type parameter* of `Iter`. Here `Dfs_iterator` will return a sequence of `Nodes`.

(3) The values returned are those passed as arguments to `yield`, just as in CLU.

(3b) [This step does not correspond to anything in CLU.] The function that should be called to `yield` all the values in the sequence is named `yielder`. This pure virtual function, declared in `Iter`, must be defined by the iterator.

(4) The user of an iterator retrieves the sequence of returned values with the C++ iterator looping construct shown earlier.

```

Dfs_iterator i(G);
Node n;
while (i.next(n))
    // do something with n

```

The way we will implement this is just as in CLU: by running the user code and the iterator code as coroutines. Each call to `next` will cause a context switch to `yielder`; each `yield` in the iterator will cause a context switch back to the user, with the yielded value as the value of `next`'s argument. When the iterator returns, `next` will return zero.

<sup>4</sup>For a simpler example, see Section 7 below.

This solution is, except for syntactic difference, identical to the CLU solution. But notice it requires no language extensions!<sup>5</sup> In the next section, we present our implementation of `Iter`.

## 7 A Simpler Example; Efficiency

Before presenting the implementation of `Iter`, here is a simpler example. The following iterator yields all the characters in a given string from left to right:

```
class Chars_in_string : public Iter<char> {
public:
    Chars_in_string(const char* s_) : s(s_) {}
private:
    const char* s;
    void yielder() {
        while (*s != '\0')
            yield(*s++);
    }
};
```

Because this is a very simple, non-recursive iterator, we could have just as easily implemented its `next` function directly.

```
// direct implementation
class Chars_in_string {
public:
    Chars_in_string(const char* s_) : s(s_), current_pos(s_) {}
    int next(char& c) {
        if (*current_pos != '\0') {
            c = *current_pos++;
            return 1;
        }
        return 0;
    }
private:
    const char* s;
    const char* current_pos;
};
```

Either of the above versions of the iterator can be used as follows:

```
void show_chars_in_string(const char* s) {
    Chars_in_string i(s);
    char c;
    while (i.next(c))
        cout << c << '\n';
}
```

---

<sup>5</sup> Well..., this is actually a slight fib. Any facility for doing coroutine-style programming is, technically, an extension to C++. However, we consider *that* to be less of an extension than an extension to the language proper.

In practice, programmers should use the direct implementation, because it avoids all the context switching overhead in the `Iter` implementation. For example, running on a Sun 3/60 using the implementation of `Iter` presented below, and a slightly tuned version of the AT&T C++ task library[8], the `Iter` version of `Chars_in_string` shown above runs between 65 and 190 times slower than the direct version.

The reason for the spread (65–190) is because the shorter the string being iterated over, the larger the slowdown. In short strings, a larger percentage of time is spent simply constructing and destructing the iterator thread, rather than actually getting the characters of the string. Construction and destruction of the iterator took approximately 0.7 milliseconds; getting the next character took approximately 0.3 milliseconds. Contrast this with the direct implementation, in which both of these functions are virtually free.

The implementation of `Iter` shown below is definitely not the fastest one possible. However, it is almost certainly the case that *no* implementation of `Iter` will result in an `Iter`-based iterator that is as fast as a direct implementation. Programmers should use `Iter` only when it is significantly easier to implement the iterator's `yielder` function than its `next` function, and when they are willing to take the performance hit.

## 8 The Implementation

Our implementation of the template `Iter` uses the AT&T C++ task library[8]. Any facility for multi-threaded programming could have been used; we used the task library because it was readily available. Notice that users of `Iter` do not have to understand the task library to be able to use `Iter`.

### 8.1 A review of the task library

First we quickly review the task library. The version of the library we describe is a small extension [9] to the distributed version. The extension allows programmers to derive an arbitrary number of levels of classes from the base class `Task`, spelled with a capital "T" in the extension.

In the task library, the term "task" is a synonym for "thread." To create a task, the user derives from `Task` and defines the `Main` member function.

```
class Mytask : private Task {
    int Main();
};
```

When an instance of `Mytask` is created, a new thread of execution is forked (in the thread sense, not in the Unix process sense) at the current point in the current thread of execution. This newly forked thread will begin running the `Main` function as soon as the `Mytask` object is sent a `start` message. For example,

```
main() {
    // ...
    Mytask t;    // current thread is forked
    t.start();   // t.Main() begins running in forked thread
    // ...
}
```

Each thread can be in exactly one of three states: *running*, *waiting*, or *terminated*. A waiting thread is a thread that performed a *blocking* operation on a *pending* object; it will resume running when the pending object becomes *ready*. The blocked operation (but not necessarily the blocked thread) will then complete. A terminated thread is one that has finished executing; it cannot be resumed.

There may be any number of running threads. If there are  $n$  instances of `Task` within a program, then there are  $n + 1$  extant threads (the original program is also a thread). Of course, real machines do not have infinitely many real processors, so there must be some processor scheduling. We will consider the common case of one real processor.

At any given moment exactly one of the running threads is assigned to the real processor and is actually executing instructions. Scheduling is *non-preemptive*: A thread gives up the real processor only when it must — that is, only when it does a blocking operation on a pending object. When this happens, the thread goes into the waiting state, and one of the other running threads is assigned to the real processor.

## 8.2 The implementation of `Iter`

Here is the complete declaration of the template `Iter`:

```
template <class yields>
class Iter : private Task {
public:
    Iter();
    virtual ~Iter();
    int next(yields& x);
protected:
    virtual void yielder() = 0;
    void yield(const yields& x);
private:
    void Main();
    void advance_iterator();
    Binary_semaphore client, iterator;
    yields* dest;
    int started, terminated;
};
```

The constructor, destructor, and `next` are public: they are the functions called by the user of the iterator. On the other hand, the functions `yielder` and `yield` are protected: the former must be defined by the iterator, while the latter is called by the iterator during the iteration. We make the inheritance from `Task` private, because our users should not care that `Iter` is implemented using `Task`.

How shall we implement the coroutine behavior of the client and iterator? We could use a high-level inter-process communication mechanism, such as a mailbox. This would make the implementation of `Iter` simple. Further, the buffering in a mailbox would reduce the number of context switches that occur during iteration, making iteration faster.

However, using a mailbox has two problems. First, we would have to implement mailboxes on top of the task library, which does not have mailboxes or any other high-level inter-process communication mechanism. We were not interested in implementing the most elegant `Iter`, but rather simply in showing that the template `Iter` *can* be implemented. Second, the buffering in a

mailbox, although it would make the implementation of `Iter` simpler and faster, would also make it more difficult for users to debug programs that use iterators.

For these reasons, we chose to implement `Iter` with a pair of binary semaphores[10], `client` and `iterator`. When the client needs the next value of the iteration, it will do a V operation on `iterator` to release the iterator, and a P operation on `client` to wait for the iterator to release the client. Conversely, when the iterator has yielded the next value or has terminated, it will do a V operation on `client` to release the client, and a P operation on `iterator` to wait for the client to request the next value. The implementation of the class `Binary_semaphore` using the task library is an exercise left for the reader.

In addition to the semaphores, `Iter` has three other private data members: a variable `dest`, shared by the client and iterator, specifying where the next yielded value should be assigned; and two boolean variables `started` and `terminated` specifying whether the iterator has started and terminated, respectively.

The implementations of the member functions of `Iter` are as follows.

The constructor simply constructs the base class `Task`, giving it a name, and initializes the state appropriately. We assume the binary semaphores are automatically initialized to zero.

```
template <class yields>
Iter<yields>():Iter() :
    Task("Iter"), dest(0), started(0), terminated(0) {
}
```

The function `Main` is called when the task is started (see the review of the task library above). When `Main` is called, it suspends the iterator until the client requests a value. It then calls `yielder`. When `yielder` returns (the iterator has yielded all its values), `terminated` is set to true, the client is released, and the iterator terminates.

```
template <class yields>
int Iter<yields>::Main() {
    iterator.P();
    yielder();
    terminated = 1;
    client.V();
    return 1;
}
```

The function `next` advances the iterator after saving the address of the variable to be assigned the yielded value. It then returns true if the iterator has not terminated, which will happen just if the iterator yielded a new value.

```
template <class yields>
int Iter<T>::next(yields& x) {
    dest = &x;
    advance_iterator();
    return !terminated;
}
```

The function `yield` assigns the yielded value to the appropriate place, releases the client, and suspends the iterator until `next` is called again.



```

template <class yields>
void Iter<yields>::yield(const yields& v) {
    if (dest)
        *dest = v;
    client.V();
    iterator.P();
}

```

The function `advance_iterator` first sends the iterator Task the `start` message if it hasn't already done so. Then, if the iterator has not yet terminated, it releases the iterator, and suspends the client until the iterator yields or terminates.

```

template <class yields>
void Iter<yields>::advance_iterator() {
    if (!started) {
        start();
        started = 1;
    }
    if (!terminated) {
        iterator.V();
        client.P();
    }
}

```

Finally, consider the destructor. If an `Iter` is destroyed after the iterator has terminated (that is, after the iteration has been exhausted), then we don't have to do anything. However, this will not always be the case. Consider, for example, a function searching for a particular node in a graph: When the node is found, the function will return, destroying the `Iter` before the iteration has exhausted. In such a case, the iterator must be terminated prematurely. The following code works for both cases:

```

template <class yields>
Iter<yields>::~~Iter() {
    cancel(-1);
}

```

The function `cancel` is inherited from the `Task` base class.

This completes the implementation of `Iter`.

## 9 Conclusions

Some problems look like they require a language extension, even after much thought. This appearance may be bolstered by the fact that other languages provide direct support for the given class of problems. However, continued searching can reveal an acceptable solution that uses the available facilities of the language. Although hard to discover, once discovered the solution can be easy to reuse. In this paper, we saw one example of such a class of problems, the problem of recursive iterators.

**NAME**

Iter – implement an iterator that “yields” its values

**SYNOPSIS of Iter.h**

```
#include <Task.h>

template <class yields>
class Iter {
public:
    Iter();
    virtual ~Iter();
    int next(yields&);
protected:
    virtual void yielder() = 0;
    void yield(const yields&);
};
```

**DESCRIPTION**

For some iterators, it is easier to implement a function that “yields” all the values in the iteration, rather than a function that returns the next value. `Iter` makes it easy to implement such iterators.

To implement a “yielding iterator,” the user should do the following:

1. Derive her iterator from `Iter<T>`, where `T` is the type to be returned by the iterator.
2. Implement the virtual member function `yielder`. When invoked, `yielder` should call `yield(n1)`, `yield(n2)`, `yield(n3)`, ... if `n1`, `n2`, `n3`, ... is the sequence of values to be returned.

The resulting iterator can be used as follows:

```
Myiter i(/* ... */);
T t;
while (i.next(t))
    // ...
```

The detailed semantics of the various operations are as follows:

`Iter()`; A new thread of execution is forked (the *iterator thread*), and invokes `yielder` in the current context. Execution is initially suspended at the beginning of `yielder`.

`int next(yields& y)`; If the iterator thread is suspended, returns 0 without affecting `y`. Otherwise, the iterator thread is resumed. Execution continues until either a call of `yield`, or the return of `yielder`. If `yield(val)` is called, the `val` is assigned to `y`, the iterator thread is suspended, and `next` returns 1. If the `yielder` returns, `y` is unchanged, the iterator thread is suspended, and `next` returns 0.

**WARNINGS**

Because of the overhead involved in context switching, a “yielding iterator” will be slower than one for which the `next` function is implemented directly. Hence, “yielding iterators” should only be used if `yielder` is significantly easier to implement than `next`, and the user is willing to take the execution time hit.

Because the implementation uses tasks, any program that uses “yielding iterators” must exit the program with the following line:

```
thistask->resultis(i);
```

where `i` is the program exit value.

**SEE ALSO**

`task(3C++)`

## References

- [1] J. Donahue. 1990. *Language Design*. Tutorial given at SIGPLAN '90 Conference on Programming Language Design and Implementation, June.
- [2] A. Koenig. 1991. An overriding concern. *The C++ Journal*, 1(3):12-15.
- [3] A. Aho, J. Hopcroft, and J. Ullman. 1983. *Data Structures and Algorithms*. Addison-Wesley.
- [4] A. Koenig. Personal communication.
- [5] B. Liskov. and J. Guttag. 1986. *Abstraction and Specification in Program Development*. McGraw Hill.
- [6] R. Griswold and M. Griswold. 1983. *The Icon Programming Language*. Prentice Hall.
- [7] A. Berztiss. 1988. Programming with generators. *Software Practice and Experience*, 18(1):73-81.
- [8] AT&T Bell Laboratories. 1992. *AT&T C++ Language System Release 3.1 Library Manual*.
- [9] J. Shopiro. Personal communication.
- [10] C. A. R. Hoare. 1978. Communicating sequential processes. *Communications of the ACM*, 21(8):666-677.

# Garbage Collection and Run-time Typing as a C++ Library

David Detlefs  
Digital Equipment Corporation  
Systems Research Center  
130 Lytton Ave, Palo Alto CA 94301  
detlefs@src.dec.com

June 18, 1992

## 1 Introduction

Automatic storage management, or *garbage collection*, is a feature that can ease program development and enhance program reliability. Many high-level languages other than C++ provide garbage collection. This paper proposes the use of “smart pointer” template classes as an interface for the use of garbage collection in C++. Template classes and operator overloading are techniques allowing language extension at the level of user code; I claim that using these techniques to create smart pointer classes provides a syntax for manipulating garbage-collected storage safely and conveniently. Further, the use of a smart-pointer template class offers the possibility of implementing the collector at the user-level, without requiring support from the compiler. If such a compiler-independent implementation is possible with adequate performance, then programmers can start to write code using garbage collection without waiting for language and compiler modifications. If the use of such a garbage-collection interface becomes widespread, then C++ compilation systems can be built to specially support the garbage collection interface, thereby allowing the use of collection algorithms with enhanced performance.

This paper presents such a garbage collection interface and implementation. In particular, the collection scheme has the following properties:

1. it requires no compiler support;
2. it requires no information from the programmer about the format of garbage-collected objects;
3. it has strategies to cope with bugs caused by aggressive optimizing compilers;
4. it allows the use of both automatically and explicitly managed storage in the same program; and
5. it invokes destructors on collected heap objects.<sup>1</sup>

---

<sup>1</sup>It is by no means clear that invoking destructors on collected objects is the proper approach for handling object finalization in garbage-collected C++ systems, but doing so at least allows the convenient handling of many common cases.

The “smart-pointer” class developed in this paper also supports *run-time type queries*. Such queries give programs the ability to ascertain the allocated type of an object at run-time, and is another feature offered in several high-level languages (e.g., CLU [22], Modula-3 [21].)

The discussion is organized as follows: Section 2 discusses related work; Section 3 presents the garbage-collection interface; Section 4 presents the template class framework and collection algorithm; Section 5 describes support for run-time type queries; Section 6 presents some further implementation considerations; Section 7 briefly discusses performance; Section 8 discusses areas for future work, giving special attention to problems that the current system does not solve; and Section 9 presents conclusions.

## 2 Related Work

There have been a number of garbage collection systems proposed and implemented for C++. To the best of my knowledge, all fail to provide at least one of the properties listed in Section 1.

Boehm and Weiser [5] developed a conservative mark-and-sweep collector aimed at use with C, but also suitable for use with C++. This collector has most of the properties listed above; the exception is that the collector does not invoke destructors on collected objects (property 5), and has insufficient information to do so. Another disadvantage is that every pointer-aligned field is considered to be a pointer, which may hurt collector performance and retain too much storage.

Bartlett [2] presents a mostly-copying collector for C++. In this scheme, stacks and registers are considered conservatively: in these areas, objects “pointed to” by bit patterns that might be pointers are retained but not copied. However, type information giving the offsets of pointers in heap objects allows copying of objects that are referenced only by pointers in the heap. Later work [3] casts this collection system in a generational framework. Bartlett’s collector was originally designed as part of a Scheme runtime written in C. When used in general-purpose C++ programming, Bartlett’s collector requires the user to invoke special macros indicating the presence of pointer fields in classes, violating property 2. Also, no provision is made for invoking destructors of collected objects, violating property 5. In previous work [6], I extended Bartlett’s work by modifying a compiler to produce the pointer-location information, and allowing concurrent mutator activity in the style of Appel, Ellis, and Li [19]; however, a collection strategy requiring a modified compiler violates property 1.

Ferreira [11] presents a C++ collector framework that allows explicitly managed storage and calls destructors. Collectibility is determined on a per-class basis; programmers must invoke a macro in the definition of each collected class that registers the class with the collector, violating property 2. The default collection algorithm is a conservative mark-and-sweep scheme similar to the Boehm-Weiser collector. There are a set of “supplementary rules” that the programmer can follow to increase performance; following these rules allows the use of other, often more efficient, collection algorithms. For example, if the programmer guarantees that each class provides a `scanref` method that invokes a collection procedure for each pointer in the class, then a potentially more efficient generational algorithm may be used. If any of the supplemental rules are used, they must be used consistently and correctly everywhere, or else the collector may malfunction. Ferreira modified a C++ compiler to follow the supplemental rules automatically; this version violates property 1.

Seliger [23] describes “Extended-C++”, a language extension that provides garbage collection as well as some other useful features. This extension defines a new language (implemented by a translator that produces C++ as output), and therefore violates property 1. Kuse and Kamimura [16] take a similar approach, designing their own object-oriented C extension that includes garbage collection.

Edelson and Pohl [9] present a copying collector based on smart pointers. The constructor for a smart pointer class performs *root registration*; it inserts the address of the pointer into a data structure, and the smart pointer destructor removes it. The collector traverses this data structure to obtain the accurate pointer identification necessary to enable the pointer modification required in copying collection. A major drawback of this collector is that the implementor of a garbage-collected class must implement a member function to copy and scan instances of that class, violating property 2. This collector does not maintain sufficient information to invoke destructors.

Later work by Edelson [10] points out some shortcomings in the copying collector described above, and describes an alternative mark-and-sweep collector based on pointer-indirection tables. This collector still violates property 2 by requiring programmers to implement a *mark* function in each garbage-collected class to mark recursively the targets of pointers contained in objects of that class. This collector is designed to allow asynchronous object finalization, as in Cedar [17] and Modula-2+ [18], in lieu of destructor invocation.

Wang [26] describes a collector based on smart pointers implemented with template classes. This system offers both reference-counting collection and “fake copy” collection; the latter, which is similar to generational mark-and-sweep, reclaims cyclic garbage. In this system, programmers use a distinguished constructor to declare that a particular reference should be registered as a root. Every garbage-collected class must include certain macros in its definition, violating property 2.

Kennedy [15] describes the support for garbage collection in the OATH class library. Kennedy points out some dangers associated with using smart pointers to get garbage collection, and advocates the use of *accessors*, which are essentially “smart references,” instead. Accessor classes must provide the same interfaces as the classes that they access, which imposes some burden on the programmer. The OATH collector is a simple reference-counting collector, but also offers the option of periodically invoking a more expensive collector that collects cyclic garbage. This backup collector is novel in that it uses a three-pass algorithm that involves reference count manipulations and does not require the identification of root pointers. This algorithm does require the identification of pointers within collected objects, and the paper does not specify how the collector gets that information; one must assume that the programmer supplies it, violating property 2 [12].

Ginter [12] surveys C++ garbage collectors, especially those involving smart pointers. He describes some difficulties with these approaches, and proposes language modifications to eliminate these difficulties. This paper addresses the concerns raised by Ginter without proposing language modifications.

### 3 Basic Garbage Collection Interface

This section presents the most basic form of the garbage collection interface, which takes the form of a template class `Ptr`.

Figure 1 shows the basic definition of the `Ptr` class; note that private members are elided. Every class `Ptr<T>` inherits from `PtrAny`; as we shall see in Section 5, `PtrAny` is

---

```

class PtrAny {
public:
    PtrAny();
    int operator==(const PtrAny& pa);
};

template<class T> class Ptr: public PtrAny {
public:
    Ptr(); // Default constructor; sets value to NIL.
    void New(); // Allocates a new T on the heap and makes this Ptr point to it.

    // Copy-constructor and assignment operator.
    Ptr(const Ptr<T>& pt);
    Ptr<T>& operator=(const Ptr<T>& pt);

    // Overloaded pointer operations.
    T& operator*();
    T* operator->();
};

const PtrAny PtrNil;

```

Figure 1: Ptr template class

---

somewhat analogous to `void*` for non-collected pointer types. The default constructor for `Ptr` sets the value of the `Ptr` to `NIL` (`= 0`). The member function `New` allocates a new `T` and causes the `Ptr` to refer to it. This is the only way to allocate a garbage collected object. `Ptr` defines copy constructors and overloaded assignment operators; these must at least ensure that the `Ptr` that is constructed or assigned to gets the same value as the argument; as we will see later, in some implementations these operations may also have other side-effects, such as the maintenance of reference counts. The overloaded `operator*` and `operator->` functions allow `Ptr<T>` variables to be used syntactically much as if they were of type `T*`. A deliberate exception to this rule is the omission of operators for pointer arithmetic, which interacts poorly with many collection algorithms. Finally, note that the `PtrAny` class, rather than `Ptr<T>`, provides an equality operator. This placement of the operator is essential to allow comparisons with a single `NIL` value; if the equality operator were in `Ptr<T>`, and required a `Ptr<T>&` argument, then we would need a different `NIL` value for every `T`.

`Ptr` is intended to be a complete interface for garbage collection. Given a program using explicit storage management, the main changes required to convert that program to use garbage collection are modifications of the types of `T*` variables to `Ptr<T>`.<sup>2</sup> In particular, the only modification that a programmer need make to a class containing pointers to collected objects is to use `Ptr` members to represent those pointers.

---

<sup>2</sup>Other changes may be required to work around the use of pointer arithmetic and the address-of operator in the original program, but these changes are generally straightforward.



## 4 Basic Ptr Implementation

My implementation of the `Ptr` interface has two aspects. The first is a framework that uses somewhat subtle properties of the C++ template system to allow the efficient and convenient generation of information useful to garbage collection algorithms, particularly the offsets of garbage-collected pointers within objects. This framework requires neither compiler support nor any information from the user beyond the use of `Ptr` types. The second aspect is the implementation of a particular collection algorithm that I feel is well suited for providing safe, portable C++ collection. The two aspects are in large measure separable; one could imagine using the template framework with several of the algorithms mentioned in Section 2 that required user-specification of the locations of pointers in objects, thus removing their violation of property 2. Sections 4.1 and 4.2 will present the framework in two parts, describing, respectively, a class hierarchy that allows the collector to treat all heap objects uniformly, and a system for generating type description information automatically; Section 4.3 will then present the particular collection algorithm.

### 4.1 Basic Class Hierarchy

This section presents the first part of the framework: those aspects of the implementation of `Ptr` that would be shared by any collection algorithm.

---

```
class WrapperBase {
protected:
    WrapperBase();

public:
    virtual WrapperBase() {}
    // ...Whatever other members and functions are needed by
    // the collection algorithm...
};

template<class T> class Wrapper: public WrapperBase {
    T elem;
public:
    // Allows us to specify a special allocator for collected objects.
    void* operator new(size_t st);

    Wrapper() {}
    sl // Only used by Ptr<T>.
    T* GetElem() return &elem;

    virtual ~Wrapper() {};
};
```

Figure 2: Wrapper template class

---

The first implementation detail we reveal is the existence of the class `WrapperBase` and the template class `Wrapper<T>`, shown in figure 2. The `WrapperBase` class presents a “garbage collected object” interface to the collector; it may include data members such as reference counts or mark bits, or virtual functions such as “`markMyChildren`”. Different collection algorithms may require different `WrapperBase` classes, but the important point is

that the collector treats all objects in the garbage-collected heap uniformly as instances of class `WrapperBase`.

Each `Wrapper<T>` class inherits the functionality of `WrapperBase`, and adds a `T` member; the `GetElem` operation returns the address of this component. `Wrapper<T>` overloads operator `new` to use an allocator appropriate for the collection algorithm. `Wrapper<T>` may also implement some virtual functions of `WrapperBase` in a way that depends on `T`; we will see an example of this in Section 4.3. Finally, note that both `WrapperBase` and `Wrapper<T>` have virtual destructors with empty bodies. This allows the collector to `delete` a `WrapperBase*` representing a reclaimed object, causing the invocation of the destructor for the associated `Wrapper<T>`; if class `T` has a destructor, it will be called by the `Wrapper<T>` destructor.

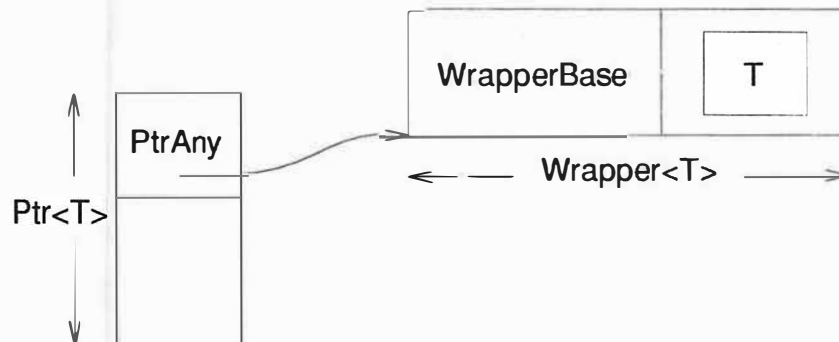


Figure 3: Relation of Ptr and Wrapper Classes

Conceptually, `Ptr<T>` contains a single member `ptr` of type `Wrapper<T>*`, and the `New` function allocates a new `Wrapper<T>` and assigns its address to `ptr`. So that `PtrAny` can be used in a similar way to `void*` (see Section 5), `ptr` is actually defined to be a protected member of `PtrAny` of type `void*`, and is cast to and from `Wrapper<T>*` by the operations of `Ptr<T>`. The “`*`” and “`->`” operators of `Ptr<T>` work by invoking the `GetElem` operation of `Wrapper<T>`. Figure 3 shows the relationships between the `PtrAny`, `Ptr`, `WrapperBase`, and `Wrapper` classes.

## 4.2 Automatic Generation of RC Maps

Most collection algorithms require a method for identifying pointers in heap objects. Many Lisp systems used *tagged data*, where one or more bits in each data word is devoted to a flag indicating the type of the value in the word. This approach requires compiler support, and has performance overhead that would not be acceptable to the C community. The method we implement uses per-type *RC maps* (reference-containing maps) to indicate which elements of aggregate types contain reference-counted pointers. This section presents the part of the framework that allows RC maps to be generated automatically, without requiring compiler support.

A number of the C++ collectors discussed in Section 2 require the user to provide information equivalent to RC maps: Bartlett’s system requires the user to invoke a macro

identifying each pointer in a struct; Ferreira's collector has a similar scheme in its non-conservative mode; Edelson and Pohl's collector requires the user to implement a copy member function for each garbage-collected class. It would obviously be more convenient and less error prone if RC maps could be generated without programmer intervention. The rest of this section presents a scheme that accomplishes this goal.

We assume the existence of an `RCMap` class providing functions for noting the offsets of `Ptr`'s within objects and for yielding these offsets in a completed `RCMap`.

---

```
class PtrAny {
    static RCMa* curMap;
    static void* containerStart;

    static void CtorInit(void* ptrStart, void*&);
    static void CtorNorm(void*, void*& ptr);

    static void SetCtorInit();
    static void SetCtorNorm();

    static void (*Ctor)(void*, void*&);
    friend class RCMapperBase;

public:
    PtrAny() { Ctor(this, ptr); }
    int operator==(const PtrAny& ra);
};
```

Figure 4: Changes to `PtrAny` for RC Maps

---

We add several private members to `PtrAny`, as shown in Figure 4. The basic idea is that the `PtrAny` class is always in one of two states, `INIT` or `NORMAL`. In the `INIT` state we are building an `RCMap` for some class *C* that may contain `Ptr`'s; a pointer to the `RCMap` being built is stored in `PtrAny::curMap`. We cause the construction of an object *O* of class *C*, and cause each `Ptr` constructed during the construction of *O* to determine its offset within *O* and add that offset to the RC map.

The `SetCtorInit` function puts `PtrAny` in the `INIT` state. The default constructor for `PtrAny` always calls the function indicated by the function pointer `Ctor`; the `SetCtorInit` function, as the name implies, sets `Ctor` to `CtorInit`. `CtorInit` simply computes the difference between the address of the `PtrAny` being constructed (passed as the first argument) and the start of the container class, which it assumes to be stored in `PtrAny::ContainerStart`.<sup>3</sup> This difference is appended to the current RC map.

`SetCtorNorm` sets the value of the function pointer `Ctor` back to `CtorNorm`, which simply does what the default constructor of `PtrAny` did in the previous version; namely, initialize the pointer to `NIL`. Having this constructor perform an indirect function call where it was inlined before obviously imposes some cost in the `NORMAL` state.

When we have built an `RCMap` for a class *T*, we store it as a static member of `Wrapper<T>`. We also create a virtual member function of `WrapperBase` that returns the RC map associated with the `Wrapper<T>` object of which the `WrapperBase` is a part. This virtual member

---

<sup>3</sup>To be more precise, only those `Ptr`'s whose addresses fall within the bounds of the original container object contribute to the RC map, since the constructor for the containing object might create other `Ptr`'s or `Ref`-containing objects at arbitrary locations.

function allows the collector to obtain RC maps for objects while still treating all objects uniformly as `WrapperBase`'s.

Next we introduce a template class `RCMapper<T>`, shown in Figure 5, to do the work of constructing an RC map for a class `T`. As we will see, constructing a single object of class `RCMapper<T>` creates an `RCMap` for type `T` and stores it as a static member of `Wrapper<T>`.

---

```
class RCMapperBase {
protected:
    // start is the start address of the current object.
    RCMapperBase(void* start);

    // This is called by the Wrapper and PtrArrayTarget special constructors
    // when they are finished.
    RCMMap* SetCtorNorm();
};

// This exists only to get the RCMMap for T built.
template<class T> class RCMapper: public RCMapperBase {
    T elem;
public:
    RCMapper() : RCMapperBase(&elem) {
        RefTarget<T>::rcMap = SetCtorNorm();
    }
};
```

---

Figure 5: `RCMapper` template class

---

The `RCMapper<T>` constructor executes after the `RCMapperBase` constructor. The body of this constructor is not shown, but its function is to set up RC map generation. Since `RCMapperBase` is a friend of `PtrAny`, its constructor can access the private members of `PtrAny` shown in Figure 4; it sets `containerStart` to the address of the `T` object in the `RCMapper<T>`, sets `curMap` to a new, empty `RCMap`, and calls `SetCtorInit` to put `PtrAny` in the `INIT` state. We now resume the `RCMapper<T>` constructor, which next invokes the constructor for its single member `elem`. If the class `T` contains any `Ptr`'s, their constructors execute; since `PtrAny` is in `INIT` mode, the offsets of these `Ptr`'s are added to `PtrAny::curMap`. When the constructor for `elem` is complete, we finally execute the body of the `RCMapper<T>` constructor, which sets `PtrAny` back to the `NORMAL` state and stores the completed RC map for class `T` in `Wrapper<T>::rcMap`.

To reach the goal of generating RC maps automatically, we must cause a static object of class `RCMapper<T>` to be initialized for each type `T` for which `Ptr<T>` occurs in the program. This is simple; we include a static member of class `RCMapper<T>` in class `Ptr<T>`. Figure 6 shows changes to the `Ptr` class to support the generation of RC maps.

With this structure, the use of a `Ptr` type such as `Ptr<Foo>` causes an `RCMap` for class `Foo` to be created and stored as a static member of `Wrapper<Foo>`. This assumes, of course, implementations that support all the template capabilities defined in the draft ANSI standard [24]; in the partial implementations that are now becoming available, it may be necessary, for example, to instantiate templates explicitly in some source file.

---

```

template<class T> class Ptr: public PtrAny {
    private:
        static RCMapper<T> rcmt;
        // ...
};

// In the implementation file for Ptr:
template<class T> RCMapper<T> Ptr<T>::rcmt;

```

Figure 6: Changes to Ptr template class for RC maps

---

### 4.3 A Particular Collection Algorithm: Deferred Reference Counting

This section presents the implementation of a particular collection algorithm within the framework described in Sections 4.1 and 4.2. The algorithm chosen is a *conservative reference-counting* collector. This algorithm has a number of desirable features for C++ collection:

- it does not move objects, avoiding a variety of problems associated with object movement in systems without compiler support;
- collection interruptions are short, allowing good interactive response;
- it is particularly simple to implement using smart pointer classes; and
- it treats pointers on the stack conservatively, avoiding a number of problems that can occur when collectors are used with aggressive optimizing compilers.

This algorithm also has some disadvantages:

- maintaining reference counts can impose considerable run-time overhead, and
- reference-counting alone cannot reclaim cyclic garbage, so a full-scale mark-and-sweep collection must be invoked occasionally.

Conservative reference counting is based on one of the two essential ideas of *deferred reference-counting* [7]. Both ideas are aimed at reducing the run-time cost of maintaining reference counts. I borrow the idea of treating stack pointers conservatively: the reference counts stored in objects count only references from heap or global objects; an object's count does not include references from the stack. When an object's reference count becomes zero, it is placed on a *Zero Count List* (ZCL) rather than deleted. The collector periodically interrupts the mutator to perform a collection. It scans the mutator stack(s) and registers to construct a *Found-On-Stack* (FOS) table which maps addresses to boolean values; if an address is found on the stack, it maps to true in this table. The collector considers each object on the ZCL: only objects not found in the FOS table are deleted.<sup>4</sup>

---

<sup>4</sup>The aspect of deferred-reference counting that I do *not* use is the idea of logging operations that change reference counts to a *transaction queue*, which is processed off-line to bring reference counts up to date. This makes sense, for example, in a multiprocessor environment where a separate processor can process the transaction queues, but probably does not gain any advantage on a current-technology uniprocessor.

The hope of conservative reference counting is that a large fraction of pointer assignments occur on the stack, thus avoiding the overhead of modifying reference counts. This strategy is usually implemented with compiler cooperation; for example, the SRC Modula-2+ system [18] used a deferred reference counting collector, and its compiler generated code to adjust reference counts only for assignments to non-stack REF (pointer-to-collected-object) variables.

Copying collectors move objects; reference-counting and (non-compacting) mark-and-sweep collectors do not. Disallowing object movement has a number of advantages for C++ collection, especially for systems that are intended to work without compiler support. To move an object, the complete and accurate set of pointers to the object must be identified; this set may include *derived pointers*, created by the compiler optimizations, that correspond to no program variable, as well as intra-object pointers used in implementing virtual base classes in multiple inheritance. Many have proposed modifying compilers to produce tables providing sufficient information to locate all pointers to objects (including, recently Diwan [8]), but this approach obviously requires a compiler closely integrated with the collector.

#### 4.4 Implementing Basic RC Collection

---

```
class WrapperBase {
    int ref_count: 30;
    int on_zcl_bit: 1;
    int mark_bit: 1;

protected:
    WrapperBase();

public:
    void Inc();
    int Dec();
    int NonZeroRC();

    void SetZCLBit();
    void ResetZCLBit();

    void SetMarkBit();
    void ResetMarkBit();

    virtual WrapperBase() {}
};
```

Figure 7: Changes to WrapperBase class for Conservative RC Collection

---

It is quite easy to implement conservative reference-counting collection in the template framework I've described. The `WrapperBase` class is revised as shown in Figure 7. We add three private data members to `WrapperBase`, encoded in one 32-bit word using bitfields: the object's reference count, a bit indicating whether it is on the Zero Count List, and a mark bit reserved for use by the "backup" mark-and-sweep collector. The `Inc`, `Dec`, and `NonZeroRC` member functions manipulate the reference counts. `Inc` increments the count, `Dec` decrements it and returns a non-zero result if the count becomes zero, and `NonZeroRC` similarly returns a non-zero result if the count is non-zero. The remaining new member

functions set and reset the ZCL and mark bits.

With these changes to `WrapperBase`, changing the `Ptr` template class to perform reference counting is straightforward. The `New` member function, the copy constructor and the overloaded assignment operator are each modified to perform as follows:

1. Check whether the current value of the `Ptr` is non-NIL.
2. If so, decrement the reference count of the original referent; if the count becomes zero, put the object on the Zero Count List and set the object's `on.zcl` bit.
3. In any case, if the new value of the `Ptr` is non-NIL, increment the count of the new referent.

To complete the collector, we modify the storage allocator to perform a collection every  $n^{\text{th}}$  allocation. As described above, a collection scans thread stacks to construct a Found-On-Stack table, then considers each object on the Zero Count List, deleting those objects whose counts remained zero and whose addresses are not found on the stack.

As described so far, the collector does not gain any performance advantage from being conservative; if `Ptr<T>` variables are used for all pointers into the heap, then all pointer assignments and initializations are reference-counted. To allow pointers on the stack to avoid reference counting, we create a variant of `Ptr<T>` called `SPtr<T>`, for “stack pointer.” An `SPtr` supports the same operations as `Ptr`, but performs no reference counting; the use of inline functions for assignment and dereference operations make the performance of an `SPtr` similar to that of a “raw pointer.” Conversion and assignment operators are defined in `SPtr` and `Ptr` in a way that make them interchangeable from the point of view of client programmers.

Since the collection algorithm includes a conservative stack scan, programmers may choose to substitute `SPtr`'s for `Ptr`'s in certain places to enhance performance. Any `Ptr` variable that the compiler will store on the stack may safely become an `SPtr`. `Ptr`'s appearing as members of classes must in general remain `Ptr`'s, since instances of the class may be allocated on the heap. There is no way (without compiler support) to check the correctness of these substitutions; programmers wanting absolute safety could use `Ptr` everywhere, but `SPtr` offers enhanced performance and maintains correctness if one follows the simple rule of using only `SPtr`'s whose storage class is `automatic`.

Using a conservative scan of the stack not only offers the possibility of enhancing performance, but also addresses many of the safety concerns that have been raised about collectors based on smart pointers. Kennedy [15] and Ginter [12] describe several scenarios which are dangerous for “smart-pointer” collectors. The dangers in most of these scenarios, particularly problems involving compiler temporaries, are avoided by using a non-copying collector that retains objects referenced by pointers found in a conservative root scan.

For example, Kennedy considers the statement

```
O2 = O1->makeCopy()->transform();
```

where `O1` and `O2` are smart pointer variables of the same type. The `makeCopy` member function makes a copy of the current object, and returns a smart pointer to the new object. The `transform` member function modifies the object to which it is applied, and returns a smart pointer to the modified object. The problem that Kennedy points out is that if the compiler generates a temporary for the result of `makeCopy`, this temporary may be the only smart pointer to the copy of `O1`. Once `operator->` is invoked on this smart pointer (returning a “dumb” pointer), the smart-pointer temporary is dead, and language rules

allow it to be destructed at any time before the end of the current scope. If it is destructed immediately, and a collection follows immediately after that, the collection may reclaim the object to which `transform` is about to be applied.

We can avoid this danger by using a conservative scan that recognizes *interior* pointers (pointers into the interiors of objects). In the `Ptr` implementation, the `T*` returned by `operator->` points into the interior of the `Wrapper<T>` that the collector considers to be the allocated object. Thus, we require that the existence of such an interior pointer on the stack suffices to prevent collection of the object into which it points.<sup>5</sup> A number of schemes have been proposed that allow interior pointers to be mapped into the addresses of containing objects; see, e.g., [5, 6, 2].

Independent of any concerns about the destruction of temporaries, it may be that some optimizing compilers will modify the pointers contained in smart pointer records in ways that violate the assumptions of the conservative scan [4]. In this case, we could provide (at some cost) a completely safe implementation along the following lines. Modify the `Ptr` and `SPtr` classes so that each contains two copies of the `ptr` member. Use one as a “write-only” copy of the pointer; it is never dereferenced, and therefore never exposed to compiler optimization. (It is declared `volatile` to prevent the writes from being optimized away.) With this approach, the collector can rely on the existence of a pointer to the head of all objects referenced from the stack. The collector would check the FOS table only for the starting address of the `WrapperBase` object. The cost of this scheme may not be as large as it seems on first consideration; memory costs are decreasing at a constant geometric rate, and the extra write costs only a single store operation per assignment, and never incurs a memory read. Section 7 presents some measurements of the cost of this scheme.

Thus, we have a range of collection options that can collect safely in the face of a variety of compiler behaviors. The safest combination, retaining objects referenced by any interior pointers found in the roots, and maintaining a “write-only copy” of the pointer in a `Ptr`, should be proof against almost all compiler behaviors. Obviously, it would be helpful if compilers were designed with garbage collection in mind; with true compiler cooperation, collectors can make stronger assumptions that improve performance.

## 4.5 Mark-and-Sweep Collection

As mentioned in Section 4.3, a disadvantage of reference-counting collection is its inability to collect cyclic garbage. The usual strategy is to periodically invoke a different collector that can reclaim cyclic garbage, such as a mark-and-sweep collector; I see no reason to depart from this strategy.

Given RC maps, it is a fairly simple matter to code a mark-and-sweep collector that treats the stacks and global data area as roots. Since we are allowing explicitly managed storage to coexist with the garbage-collected heap, we must also treat this “malloc heap” conservatively as a source of collection roots. Supporting mark-and-sweep collection places two requirements on our heap data structures:

1. We must be able to identify the allocated portion of the explicitly managed heap, so

---

<sup>5</sup>One might be tempted to optimize by observing that the pointer returned by `operator->` always points to a fixed offset from the containing `Wrapper<T>`, but statements such as

```
O2 = O1->makeCopy()->xxx.transform();
```

show that any interior pointers must be sufficient to prevent collection: here we dereference the `xxx` member of the referent of the smart pointer, which is at an unknown offset from the start of the containing `Wrapper<T>`.



that it can be scanned for possible collection roots.

2. We must be able to identify the start and extent of every object in the garbage-collected heap, both to determine which object an interior pointer points to, and to locate unreferenced objects during the sweep phase.

There are many possible organizations of the heap that allow these requirements to be satisfied; I have not yet implemented one, and will not present one here. I will note that for programs that make extensive use of explicitly managed storage, treating the explicitly managed heap as a source of conservative roots may to be expensive, so it will probably be important to invoke the mark-and-sweep collector as infrequently as possible.

## 5 Run-Time Type Information

The `Ptr` class also include another feature called *typecodes*, inspired by Modula-2+ `REF` type. Typecodes are a minimal form of run-time type information: a distinct integer is associated with each type `T` for which a `Ptr<T>` type is instantiated in the program. This integer can be found given either the type `Ptr<T>` or an instance of that type:

```
Ptr<T> pt;  
pt.New();  
if (pt.TypeCode() == Ptr<T>::StatTypeCode()) { ... }
```

What makes this capability useful is that any `Ptr<T>` can be converted into a `PtrAny`, and `PtrAny` is defined so that one can still query the type code. Further, `Ptr<T>` assignment operator for `PtrAny` arguments allows the assignment not only of `PtrNil` to a `Ptr<T>`, but also of non-NIL `PtrAny` values whose typecodes indicate that they are really `Ptr<T>`'s. If the typecode of the `PtrAny` does not match the typecode of `Ptr<T>`, the copy constructor and assignment operator raise an exception. The copy constructor for `PtrAny` arguments is defined in the same way.

While in general it is a good idea to search for ways of substituting statically-type-checked mechanisms such as virtual functions or template classes for run-time type discrimination, the technique remains useful in some circumstances. For example, run-time type queries enable the use of heterogeneous collections; in my system, such collections would have element type `PtrAny`. Another example occurs when a virtual function's implementation would differ only slightly in derived classes; it may be more convenient to implement the function once in the base class, using run-time type queries to produce the appropriate subclass-dependent behavior.

A number of systems have been proposed for adding run-time type information to C++. Gorlen's NIHLib [13] provides the functionality of typecodes and more; programmers may query whether an object is derived from any type, and acquire the name of the an object's actual type as a string. To offer this functionality, all classes must be derived from a distinguished base class `Object`, and class implementors must manually provide the information used by the run-time mechanism. The ET++ user-interface framework [1] provides similar capabilities. Interrante and Linton [14] propose a `Dossier` class as a standard interface for class information; the functionality provided by this interface subsumes the functionality of typecodes. Interrante and Linton propose extending the language to generate a virtual function automatically for each class that returns an appropriate `Dossier` object; in the absence of such a language extension, however, programmers must follow the convention of

implementing such a virtual function manually in each class. Lenkov, Mehta and Unni [20] also propose a set of language extensions that would enable run-time type queries.

Again, I emphasize that the mechanism proposed here is quite minimal; features such as returning the class name as a string, or querying whether the actual type of a `PtrAny` is some subtype of a given static type, are not supported. The main strength of this proposal is that it does not require

- any extension to the language definition or compilers;
- a common base class shared by all classes; or
- additional code in every class supplied explicitly by programmers.

In contrast, each of the proposals above requires one or more of these properties.

The template framework we have defined for collection makes implementing typecodes quite simple. `TypeCode` is a simple class with a single public integer member; the constructor of `TypeCode` guarantees that each `TypeCode` object has a distinct value for this member. `Typecode` ensures this property by using a static member as a monotonically increasing counter. Every `Wrapper<T>` class has a static `TypeCode` member whose initializer invokes this constructor.

## 6 Implementation Considerations

This section presents implementation details that would have complicated the previous discussion.

### 6.1 Dynamically-Sized Arrays

The program fragment

```
int* ip = new int[j];
```

allocates an array of integers whose size is determined at run-time by the value of the variable `j`. It is not possible to use the `Ptr` class to allocate a dynamically-sized, garbage-collected array. To allow the use of such arrays, I provide a class similar to `Ptr` called `PtrArray`.

For the most part, `PtrArray`'s are identical to `Ptr`'s. The main differences are:

- The `New` operation of `PtrArray` takes an integer parameter telling how many elements will be in the newly-allocated array.
- `PtrArray` does not provide overloaded pointer operations. Instead, the only mechanism it provides for accessing its contents is an overloaded operator `[]`, which provides bounds-checked access to the array elements.<sup>6</sup>
- `RCMap` is modified slightly to work more efficiently with `RefArray`: an `RCMap` also includes the number of bytes between the last `Ptr` in an object and the beginning of the next object when the object is included in an array. This information is generated with the same automatic mechanism used before.

<sup>6</sup>Bounds-checking can be turned off by setting a preprocessor variable.

## 6.2 Circular Definitions

When using pointers, it is desirable to be able to write definitions such as:

```
struct IntList {
    int i;
    IntList* next;
};
```

If `Ptr`'s are to be as powerful as pointers, one should be able to write

```
struct IntList {
    int i;
    Ptr<IntList> next;
};
```

This generates the constraint that it must be possible to instantiate at least the declaration of `Ptr<T>` when `T` is an incomplete type. (Recall that the *declaration* of a class gives its signature; the *definition* its implementation.) Thus, the result of instantiating the declaration of `Ptr<T>` may not include any classes that have elements of type `T`. Note that this constraint does not apply to the definition of `Ptr<T>`, which may use `T` freely.

Two “tricks” are used to avoid using `T` in the declaration of `Ptr<T>`. The first trick has already been mentioned: the `ptr` member of `PtrAny` is declared as a `void*`, but treated as a `Wrapper<T>*` in the bodies of the implementations of `Ptr<T>` via casting. The second trick involves the `RCMapper<T>` type used to generate RC maps automatically. Section 4.2 stated that every class `Ptr<T>` includes a static member of type `RCMapper<T>`. This is not actually true, because `RCMapper<T>` defines a member of type `T`, which would cause instantiations with incomplete types to give a compile-time error. Instead, `Ptr<T>` has a static member whose type is `RCMapperIndirect<T>*`. Nothing is revealed about the class `RCMapperIndirect<T>` in the declaration of `Ptr`. The implementation reveals that `RCMapperIndirect<T>` has a static member of type `RCMapper<T>`; initializing this member creates the RC map for type `T`.

## 6.3 Complete Class Hierarchy

We started with class `Ptr`, and added a number of related classes, as follows:

- `SPtr` allows programmers to get pointer-like efficiency for stack-allocated `Ptr`'s; objects referenced by these stack pointers will be retained because of the conservative stack scan used by the collector.
- `PtrAny` enabled run-time type queries and heterogeneous collections.
- `PtrArray` supported garbage-collected references to arrays of objects.

Figure 8 shows the complete class hierarchy associated with `Ptr`'s. The new features introduced by this figure are described below.

- The pair of classes `PtrAny` and `SPtrAny` are related in the same way as `Ptr` and `SPtr`, which we have already discussed. `PtrArray` and `SPtrArray` also have this relationship.
- `BPtr<T>` abstracts the functionality common to `Ptr<T>` and `SPtr<T>`, such as the dereference operators. (Read the “B” as “Base.”) Similarly, `BPtrAny` factors out functionality common to both `PtrAny` and `SPtrAny`, such as the `typeCode` virtual function, and `BPtrArray` defines the functionality common to `PtrArray` and `SPtrArray`, such as the overloaded operator[].

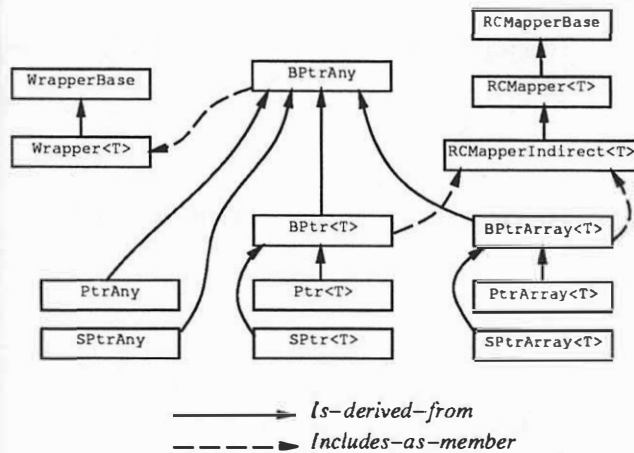


Figure 8: Garbage Collector Class Hierarchy

## 7 Performance

I have modified two non-trivial programs to use the garbage collection system described in this paper. One, GROBNER, is an implementation of the Grobner basis algorithm [25], a theorem-proving application. The other, HYPER, is a simulation of a hypercube architecture, done by Donald Lindsey at Carnegie-Mellon University.

	GROBNER	HYPER
1 ET-no-GC	4.4 sec	10.4 sec
2 ET-GC (/ ET-no-GC)	8.8 sec (2.00)	16.5 (1.59)
3 # Allocations	163301	11334
4 # RC ops	515735	5496586
5 On stack (%)	188783 (37%)	2873609 (52%)
6 Non-stack (%)	326952 (63%)	2622977 (48%)
7 # Collections	18	3
8 Collect Time	2.11 sec	0.09 sec
9 Avg. collect time	0.11 sec	0.03 sec
10 Max. collect time	0.17 sec	0.08 sec

Figure 9: Performance of Reference-Counting Garbage Collection

Figure 9 shows measurements of these programs running on representative input files. All measurements were taken on a MIPS R3000-based DECStation 5000 with 96 MByte of memory, running Ultrix. Memory size was sufficient to prevent paging in all tests. "ET-

no-GC” shows the elapsed time of the program using explicit storage management. The numbers shown in this line and all others are the averages over 10 runs. “ET-GC” shows the elapsed time of the same program using garbage collection, and includes the ratio with the first line. “# Allocations” indicates the number of allocations performed during the run. “# RC ops” is the number of *reference-counting operations* that occurred in the run; these operations are the assignment operators and copy constructors of `Ptr` and `SPtr` types. The “On stack” line indicates the number of these operations that occurred on `SPtr`’s, thus avoiding reference counting; the remainder are shown in the “Non-stack” line. The “# Collections” line shows the number of collections that occurred in the run. The last three lines show the total time spent in collections, and the average and maximum durations of collection interruptions.

The overall performance of the garbage collection algorithm in these tests is somewhat disappointing, as shown by the increase in overall elapsed time. Section 4.3 stated that a hope of conservative reference counting is that the majority of pointer assignments occur on the stack, and are therefore not reference counted; lines 3-6 show that this hope is not fulfilled in the programs tested. Two programs is still a small sample size; further measurements are obviously desirable.

One positive aspect of these measurements is that the interruptions due to conservative reference-counting collections are quite short, as shown in lines 9 and 10. Thus, the cost of collection can be smoothly distributed, which would be an advantage in interactive applications. The number of allocations between collections was set at 10,000 entries in these tests, but can be modified by the user. Increasing this number to 100,000 for GROBNER decreased the number of collections to 3, and decreased total collection time by 19%. However, the length of the average collection interruption increased from 0.11 sec to 0.57 sec.

A final observation is that using the two-pointer scheme described in Section 4 for enhanced safety does not significantly increase the garbage collection overhead for these programs. Using this scheme for GROBNER gave no detectable difference in the overall elapsed time; for HYPER, elapsed time increased by 5%.

## 8 Future Work

This section examines opportunities for future work in this area. Some opportunities are made available by strengths of this collection interface; others, unfortunately, come out of weaknesses.

### 8.1 Strengths: Other Collection Algorithms

A strength of this collection interface is that it should admit several implementations. In fact, other collection algorithms should be able to re-use much of the template framework presented in Section 4. For example, one can easily imagine implementing the `Ptr<T>` interface with Bartlett’s mostly-copying algorithm instead of reference counting. Using the `Ptr<T>` syntax for pointers to garbage-collected objects would enable the coexistence of garbage-collected and explicitly managed storage in Bartlett’s algorithm.<sup>7</sup> The main advantage gained would be the automatic generation of RC maps, avoiding the requirement that programmers provide equivalent information explicitly.

<sup>7</sup>Note that the explicitly managed heap would have to be treated as part of the conservative root set, as noted in Section 4.5.

## 8.2 Weaknesses

The current interface has several weaknesses. Solutions to these problems would be interesting future work.

One weakness is that a class `T` used to instantiate `Ptr<T>` must have a constructor of no arguments if it has any constructors. If `T` has any constructors, the constructor of no arguments is the only one that can initialize a `T` object allocated using `Ptr<T>::New`. This requirement is at best inconvenient; for some types, it is worse, since it may not be possible to sensibly define a constructor of no arguments for the type. (Consider, for example, something like `NonEmptySet`.)

Another weakness of the interface concerns “casting up,” or *widening*. Casting up is a common, natural, and safe C++ idiom:

```
class A { ... };

class B : public A { ... };

B* bp = new B;
A* ap = bp;           // Implicitly casts bp to A*.
```

If `Ptr`'s are to be semantically equivalent to pointers, there should be a similar idiom involving `Ptr`'s; we should be able to convert a `Ptr<B>` to a `Ptr<A>`. Ideally, this would be done with the same syntax used by pointers.

The best mechanism I have been able to devise for casting up `Ptr`'s uses a template class `Widen<From, To>`. The constructor for `Widen` takes a `Ptr<From>&` argument; `Widen` provides a single operation `ToPtr` that returns a `Ptr<To>`. Using `Ptr`'s and `Widen`, the example above would be rewritten as

```
Ptr<B> bp;
bp.New();
Ptr<A> ap = Widen<B, A>(bp).ToPtr();           // Convert bp to Ptr<A>.
```

`Widen` is guaranteed to give a compile-time error message if `To` is not a base class of `From`. The current version of `Widen` works only with single inheritance hierarchies implemented in the usual way, where casting up a pointer preserves the pointer value. I have a design for a version of `Widen` that would work with multiple inheritance, but would require the `Ptr` and `SPtr` classes to contain two pointers instead of one. This scheme could be combined with the two-pointer scheme described for added safety in Section 4.4.

Another capability that would be convenient but is not supported by the current system is the capability to “cast down” (or *narrow*) to a legal type other than the allocation type of an object. That is, continuing the example above, if one had a `PtrAny` whose allocated type is `Ref<B>`, one would like to be able to query whether the `PtrAny` is also of type `Ptr<A>`, and convert the `PtrAny` to that type if the answer is `true`. The system presented in this paper provides no support for this operation. It may be that many future compilers will generate run-time data structures containing information sufficient to support these queries in order to implement the C++ exception model; if so, it may be useful to extend the C++ language to make some of this information available to programmers [20]. In any case, C++ presently offers no guarantees for the results of casting down; the situation is no worse for in the system I propose.

## 9 Conclusions

This paper has presented a smart pointer template class interface to garbage collected storage, and has argued that such an interface may be sufficiently convenient to be used even

if language changes are made to support collection. It also presented a compiler-independent implementation of this interface. This implementation was divided into a “framework” that could support a number of different collection algorithms, and a particular collector based on conservative reference counting. While the performance of this implementation is not especially exciting, its existence allows the interface to be used without any special compiler support. If the interface (or one like it) is found to be useful, gains popularity, and is later standardized, one could imagine compilation systems giving special treatment to the `Ptr` class that would enable more efficient collection algorithms. This is a potential scenario for the wide-spread adoption of garbage-collection by C++ programmers.

## References

- [1] Gamma; Andre Weinand; Erich and Rudolf Marty. ET++ – an object-oriented application framework in C++. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 46–57, New York, NY, 1988. ACM.
- [2] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, February 1988.
- [3] Joel F. Bartlett. Mostly-copying collection picks up generations and C++. Technical Report TN-12, Digital Equipment Corporation Western Research Laboratory, October 1989.
- [4] Hans-Juergen Boehm. Simple GC-safe compilation. Position paper in 1991 OOPSLA Garbage-Collection Workshop, 1991.
- [5] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.
- [6] David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie-Mellon University, May 1990.
- [7] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.
- [8] Amer Diwan. Stack tracing in a statically typed language. Position paper in 1991 OOPSLA Garbage-Collection Workshop, 1991.
- [9] Daniel Edelson and Ira Pohl. A copying collector for C++. In *Usenix C++ Conference Proceedings*, pages 85–102, Berkeley, CA, 1991. Usenix Association.
- [10] Daniel R. Edelson. A mark-and-sweep collector for C++. In *Proceedings of the ACM Conference on Principles of Programming Languages*, New York, NY, 1992. ACM. Edelson has made a longer version of this paper available electronically.
- [11] Paulo Ferreira. Garbage collection in C++. Position paper in 1991 OOPSLA Garbage-Collection Workshop, 1991.
- [12] Andrew Ginter. Cooperative garbage collectors using smart pointers in the C++ programming language. Master’s thesis, University of Calgary, December 1991.

- [13] K. E. Gorlen. An object-oriented class library for C++ programs. *Software Practice and Experience*, 17(12):899–922, December 1987.
- [14] John A. Interrante and Mark A. Linton. Runtime access to type information in C++. In *Usenix C++ Conference Proceedings*, pages 233–240, Berkeley, CA, 1990. Usenix Association.
- [15] Brian Kennedy. The features of the Object-oriented Abstract Type Hierarchy (OATH). In *Usenix C++ Conference Proceedings*, pages 41–50, Berkeley, CA, 1991. Usenix Association.
- [16] Kazushi Kuse and Tsutomu Kamimura. Generational garbage collection for C-based object-oriented languages. Position paper in 1991 OOPSLA Garbage-Collection Workshop, 1991.
- [17] Butler W. Lampson. A description of the Cedar language; a Cedar language reference manual. Technical Report CSL-83-15, Xerox PARC, 1983.
- [18] Paul Rovner; Roy Levin; and John Wick. On Extending Modula-2 For Building Large, Integrated Systems. Research Report 3, Digital Equipment Corporation Systems Research Center, 1985.
- [19] John R. Ellis; Kai Li; and Andrew W. Appel. Real-time Concurrent Collection on Stock Multiprocessors. Research Report 25, Digital Equipment Corporation Systems Research Center, February 1988.
- [20] Dmitry Lenkov; Michey Mehta; and Shankar Unni. Type identification in C++. In *Usenix C++ Conference Proceedings*, pages 103–118, Berkeley, CA, 1991. Usenix Association.
- [21] Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [22] B. Liskov; R. Atkinson; T. Bloom; E. Moss; C. Schaffert; R. Scheifler; and A. Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.
- [23] Robert Seliger. Extending C++ to support remote procedure call, concurrency, exception handling, and garbage collection. In *Usenix C++ Conference Proceedings*, pages 241–264, Berkeley, CA, 1990. Usenix Association.
- [24] Bjarne Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley, Reading, Massachusetts, 1991.
- [25] J. P. Vidal. The computation of Grobner bases on a shared memory multiprocessor. Technical Report CMU-CS-90-163, Carnegie Mellon University, August 1990.
- [26] Thomas Wang. The MM garbage collector for C++. Master's thesis, California Polytechnic State University, San Luis Obispo, October 1989.



# Encapsulating a C++ Library

Mark A. Linton

*Silicon Graphics Computer Systems*

## Abstract

Encapsulation is the hiding of internal details from the user of an abstract data type, class or module. Encapsulating a class library requires more than combining a set of classes that are encapsulated individually. Libraries need to hide the details of how objects are created because some kinds of objects may be represented by composites as opposed to single instances. Implementation classes and members also must be hidden from the user of a library, even if this hiding conflicts with the user's desire to reuse library code.

In this paper, we present the encapsulation techniques used in the InterViews 3.1, a C++ class library for building user interfaces. These techniques have been formulated from the experience of building and releasing InterViews over a period of several years.

## 1 Introduction

Building and maintaining a class library is more difficult than building and maintaining an application because the number of external interfaces is much larger. For example, consider a library of 20 classes, with an average of 5 public functions per class and 20 lines of code per function. This library has 100 external interfaces for 2,000 lines of code. In contrast, a 2,000-line application would typically have closer to 10 or 20 external interfaces. This example matches our actual experience with the InterViews class library[2] and the Dbx debugger[5]. Both InterViews and Dbx are about 25,000 lines of code, but InterViews has an order of magnitude more external interfaces.

To reduce the complexity of building and maintaining a class library, we must encapsulate as much of the library implementation as possible. Encapsulating a library is more than encapsulating each class individually. In particular, we need to hide the details of object creation to give the implementation the freedom to create an instance from a library class, clone an existing instance, or create a composite of several instances from several classes.

In this paper, we present three encapsulation techniques that we have applied in version 3.1 of the InterViews class library. The first technique is to create a layer of abstraction for object creation and has the most significant effect on users of the library. The second technique is to hide implementation-oriented class members even if the members could be useful to library user. The third technique is the use of implementation classes to hide external dependencies. We also discuss the effect of these library encapsulation techniques on documentation.

## 2 Object creation

The primary motivation for abstracting library object creation is to allow the implementation to change the way an object is instantiated without effecting the library user. For example, suppose the user wishes to write code to create a push button object. The library could implement a push button class, or a push button could be created by composing an object that implements button input behavior with an object that implements the output behavior associated with a push button.

The result of abstracting object creation is that library users *do not call constructors*. Instead, a library user calls a function to create an object. This function could be a global or static member function; however, we use a virtual member function on another class. This class contains functions for creating a variety of related kinds of objects. We call a creator class a “kit”; such a class is also sometimes called an “object factory”.

In the push button example, using a push button class one might write the following code to create an instance:

```
Button* b = new PushButton("hi mom!", button_callback);
```

Using a kit, one instead would write

```
Button* b = kit.push_button("hi mom!", button_callback);
```

A critical assumption here is that a PushButton class adds no protocol beyond what is supported by the Button class. From the library designer’s perspective, the existence of any class that does not define new protocol is an implementation decision and should be hidden from the library user.

Kits can help organize a library for the user, providing a higher-level structure to the classes. For example, InterViews 3.1 provides kits for stylistic components, such as buttons and menus, and layout components, such as boxes and glue for document formatting. We intend to add more kits in the future, but we anticipate the number of kits will be quite small (5-10) compared to the number of classes (around 100).

Kits also have four benefits with respect to class definitions:

- Classes provided for convenient construction can be eliminated.
- Classes that add no protocol can be hidden.
- Constructor overloading and default parameters can be avoided.
- Accessing existing objects is simplified.

In the remainder of this section, we use examples from InterViews to demonstrate the value of kits. We also show how making the kit member functions virtual allows us to provide alternate implementations of a kit.

## 2.1 Eliminating classes

An important feature of InterViews is high-level support for sophisticated layout. In particular, we have extended the TeX “boxes and glue” formatting model[4] from static pages to all user interface components. TeX boxes and glue have a horizontal or vertical orientation. For example, a “vbox” arranges components top-to-bottom and “vglue” has a specified height and vertical stretchability or shrinkability.

A user of the InterViews library would like to be able to create a vglue object. Our original approach was to define a VGlue class in C++, which derived from a base Glue class. For convenience, we provided both a VGlue constructor that specified only the natural size (assuming infinite stretchability) and a constructor that defined the natural size, stretchability, and shrinkability.

The VGlue class added no protocol to Glue; indeed the only member functions defined were the constructors. Furthermore, the implementation of the constructors simply passed the appropriate parameters to the parent Glue constructor.

InterViews 3.1 provides a LayoutKit class that has an overloaded set of vglue member functions. The functions return simply a pointer to a *glyph*, which is the base class for objects that are allocated screen space.

Using the layout kit, there is no longer any need for the VGlue class at all, as the layout kit member function can call the Glue constructor directly. Figure 1 shows the VGlue constructor code and corresponding layout kit member functions.

An unexpected effect is that the kit approach is *more* efficient than the class approach. The runtime performance is equivalent. Although the kit call has an extra level of indirection over a direct constructor call because the vglue member function is virtual, the VGlue constructor must do some work to set up the virtual function table. Both approaches call the Glue constructor.

The savings from the kit approach is the code size necessary to define the VGlue constructors and the VGlue virtual function table. A knowledgeable compiler might be able to eliminate the VGlue code and table, but this optimization comes for free when using a kit.

---

```
VGlue::VGlue() : Glue() { }
VGlue::VGlue(Coord natural) : Glue(Dimension_Y, natural, fil, 0.0, 0.0) { }
VGlue::VGlue(
    Coord natural, Coord stretch, Coord shrink
) : Glue(Dimension_Y, natural, stretch, shrink, 0.0) { }

Glyph* LayoutKit::vglue() { return vglue(0); }
Glyph* LayoutKit::vglue(Coord natural) { return vglue(natural, fil, 0); }
Glyph* LayoutKit::vglue(Coord natural, Coord stretch, Coord shrink) {
    return new Glue(Dimension_Y, natural, stretch, shrink, 0.0);
}
```

---

Figure 1: VGlue class and layout kit implementations

---

Introducing the LayoutKit class allowed us to remove 25 classes from the InterViews library. Figure 2 shows the effect on the class library diagrammatically, showing the old class hierarchy with obsolete classes in italics.

On the other hand, the kit approach could continue to use a VGlue class if there were some execution or memory usage benefits. For example, LayoutKit::vglue() could return an instance of a special class that does not store any size information—it simply returns fil (infinite) stretchability and zero for everything else. The important idea is that the kit isolates the library user from the presence or absence of the VGlue class.

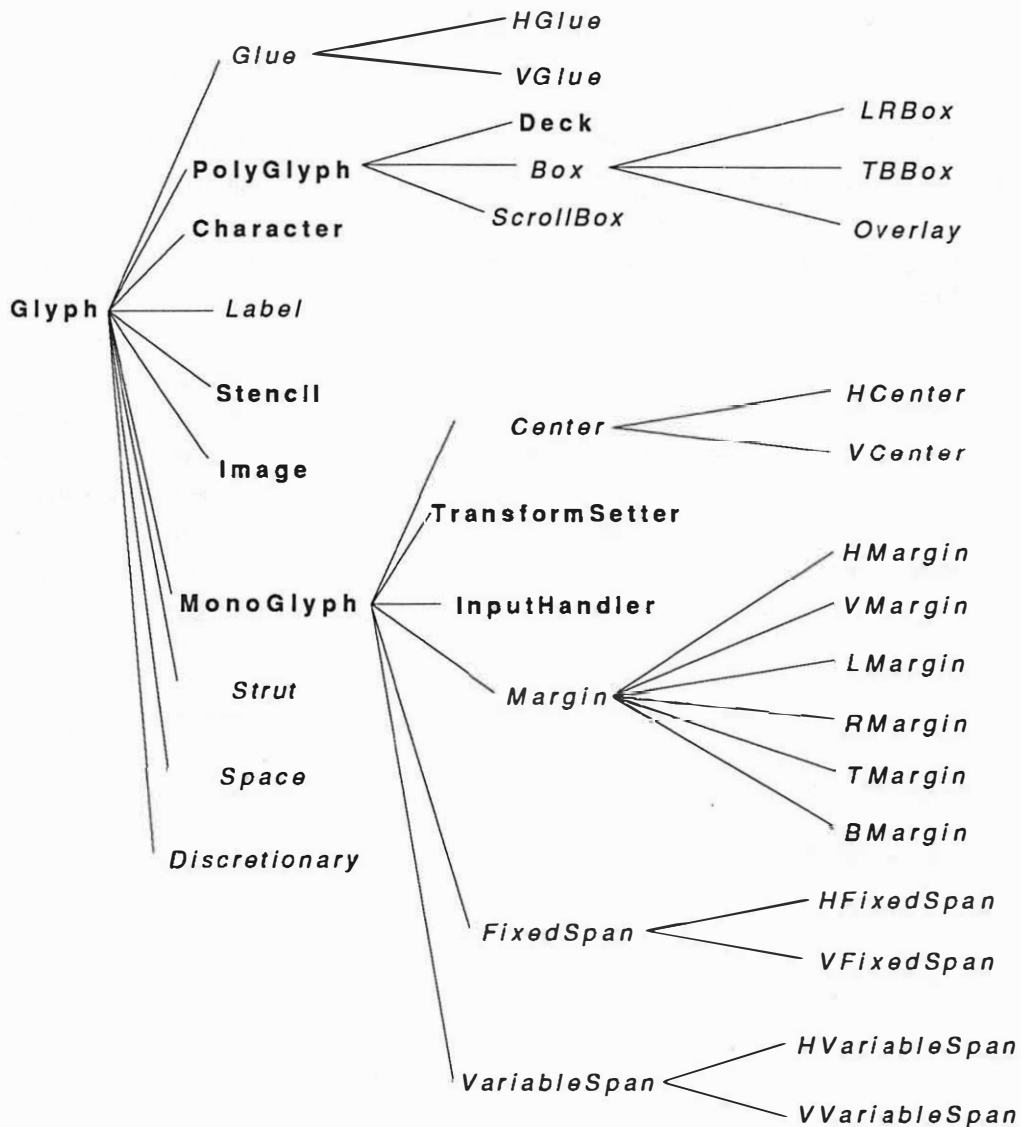


Figure 2: Layout classes

## 2.2 Hiding classes

Even when the use of a kit cannot eliminate a class, the class may become irrelevant to the library user. In this case, the library developer need not support or document the class. An example from the InterViews layout kit is the discretionary class. In TeX, a discretionary is an object that defines a “penalty” for generating a line break and a potentially different appearance in the case that a line break occurs. For example, a paragraph break is a “good” place to break, white space is an “ok” place to break, and a hyphenation point between characters in a word is a relatively “bad” place to break. If a break occurs on white space in justified text, then the white space becomes zero-width.

The LayoutKit class provides member functions for creating discretionaries. The glyph base class defines a *compose* member function that returns a potentially different glyph in the case of a break. The LayoutKit implementation of discretionaries therefore must use a new glyph subclass that implements compose; however, this subclass need not be visible to the library user.

The key characteristic of classes that can be hidden is that they do not add any protocol beyond their base class. Whether a class adds protocol often depends on whether associated state is editable. For example, we can eliminate Glue because there are no operations to change the stretchability of an existing glue object. An application will create a new glue object instead of changing an existing one. In contrast, we cannot eliminate TransformSetter because we want to be able to modify its transformation matrix without creating a new object.

The disadvantage of hiding classes is that users cannot use subclassing to reuse the behavior of these classes. The library designer must compare the cost of maintaining a class that is a public part of the library against the value of subclassing. In the case of the Discretionary class, the implementation is so simple (under 100 lines of code) that the subclassing value is negligible.

## 2.3 Defining constructors

In C++, kit member functions have an advantage over constructors in that they can use different names when a call might otherwise be ambiguous. Before defining a layout kit for InterViews, this problem arose when we wanted to be able to fix the size of an object in either one or both dimensions. We had defined the following class:

```
class FixedSpan : public MonoGlyph {
public:
    FixedSpan(Glyph*, DimensionName, Coord span);
    FixedSpan(Glyph*, Coord x_span, Coord y_span);
    // other functions
};
```

DimensionName is an enumerated type with values Dimension\_X and Dimension\_Y. We found that some compilers could not resolve the calls

```
FixedSpan(g, Dimension_X, 10.0),
FixedSpan(g, 10.0, 10.0)
```

because of the possibility of an implicit conversion from an integer to a float. The only solution with a class-based approach (other than forcing the user to put in a cast) is to define additional

classes, such as `FixedXSpan` and `FixedYSpan`. Using `LayoutKit`, we can simply define member functions with different names when overloading resolution is not sufficient:

```
Glyph* fixed_span(Glyph*, Coord x_span, Coord y_span);  
Glyph* fixed_span_dimension(Glyph*, DimensionName, Coord span);
```

A kit can maintain state that must be passed to constructors but is inconvenient for users to pass through parameters. We use this ability to maintain a current style in the kit that creates buttons, menus, and scrollbars. However, this state could also be used to specify what location parameter to pass to *operator new* without requiring that all object creation calls be aware of a location option.

Finally, kits also avoid ambiguity problems associated with default parameters. Because the user creates objects solely through a kit, the need for constructors with parameters is reduced to the case where an object or some data associated with an object is immutable.

## 2.4 Extensible modules

Kits are essentially modules—classes with a single instance in C++. We could implement a kit as a class with static member functions, but by making the member functions virtual we can make it possible to choose from several implementations of a kit at runtime.

For example, the `InterViews 3.1 WidgetKit` defines functions for creating common user interface components such as buttons, menus, and scrollbars. `WidgetKit` is an abstract base class; subclasses create objects that support a particular look-and-feel. For example, the `MFKit` implements a subset of the `OSF/Motif` look-and-feel. Similarly, an `OLKit` could implement the `OpenLook` look-and-feel. This approach to supporting multiple styles is similar to the functionality of the `Solbourne OI toolkit`[1], though `OI` uses global function calls instead of virtual calls on a kit class.

`WidgetKit` defines a single static member function called *instance* that returns a pointer to the kit object. The first time this function is called, it will create the object and store a pointer to it in a static data member. Subsequent calls simply return the stored pointer. `WidgetKit::instance` creates the appropriate subclass depending on a user or system-defined property.

A kit also normally defines a protected member function to assign the kit instance pointer directly. Thus, a user can define a subclass, redefine any virtual functions as desired, and create an instance of that class directly before the kit is first accessed.

This technique is applicable to other one-of-a-kind objects. The `InterViews Dispatcher` class, which routes input and timer events, also defines a static member function for returning the instance and a function for setting the instance. All other dispatcher functions are virtual, allowing the library user to redefine the behavior of the dispatcher if so desired.

## 2.5 Accessing objects

For objects such as colors and fonts, application code often wants to access an existing object instead of always creating a new one. For example, one might wish to find the color named “red”. If the name is known but a color object has not yet been created, the object should be created. If the name is unrecognized, then one should not receive a valid object.

This functionality does not lend itself intuitively to using constructors because the caller is not creating an object so much as looking for an object. Constructors also cannot easily return an invalid object. Kits provide the opportunity to lookup an object by name, create one if necessary, and return a nil pointer if the object cannot be created.

### 3 Hiding class members

Encapsulation of an individual class involves a tradeoff between what is exposed for potential reuse and what is hidden for potential future change. In an application, the use of a class is often limited and the cost of change relatively low. In contrast, a class in a library has much more widespread use and the cost of changing a public or protected interface is much greater. This dichotomy causes the library user, who is developing on an application, to want to reuse as much of the library implementation as possible. The library implementor, on the other hand, wants to hide as much as possible.

In the past, InterViews exposed implementation through both protected data and functions. Our current strategy is to make *all* data members private and to make the “default” access to functions be private. We treat “over-protection”—access that is private when it should be public or protected—as an enhancement request that is straightforward to provide. The opposite—access that should be changed to private—cannot be achieved without the possibility of breaking existing code.

#### 3.1 Data members

Making data members private gives the implementation freedom in moving data to another object or in changing a representation. Access functions for reading and writing allow users to store and retrieve the data.

An example from InterViews demonstrates the benefits of using access functions. The Telltale class defines an appearance that depends on a set of flags, including whether the telltale is disabled or enabled, highlighted or not, and chosen or not. The first Telltale implementation stored each flag in a separate bit field data member of the Telltale object. Access functions were provided to test the current state of the Telltale, such as whether it was enabled. Figure 3 shows the original Telltale class definition.

In the most recent implementation, the flags are stored in a separate object so that the state can be shared among several objects. The Telltale access functions have been modified to access the state indirectly, and code that uses the original Telltale interface continues to work without modification. Figure 4 shows the new Telltale class definition and the implementation of the *enabled* functions.

#### 3.2 Protected vs. private

Given the data members of a class are private, one still must decide whether to provide access functions for the data and whether those access functions should be public or protected. Permitting access promotes the greatest reuse, but also exposes the implementation.

---

```
class Telltale : public Glyph {
public:
    // constructors, other public functions
    void enabled(boolean);
    boolean enabled() const;
private:
    boolean enabled_ : 1;
    // other data, private functions
};
```

Figure 3: Original Telltale definition

---

---

```
class Telltale : public MonoGlyph {
public:
    // same public interface
private:
    TelltaleState* state_;
    // other data, private functions
};

void Telltale::enabled(boolean b) {
    state_>set(TelltaleState::is_enabled, true);
}

boolean Telltale::enabled() const {
    return state_>test(TelltaleState::is_enabled);
}
```

Figure 4: New Telltale definition

---

An early mistake we made was to define implementation functions as protected. From a maintenance point of view, protected is really no different from public. In either case, a class user may become dependent on the function. Furthermore, we found that protected encouraged subclassing for greater access even in the cases where instantiation would have been more appropriate.

Our current approach is to define functions as “private” by default. If the function is needed externally, then it should probably be public. We use protected primarily for constructors of abstract classes.

## 4 Implementation classes

The representation and implementation of several InterViews classes depends on the X Window System, but the interface to these classes is independent of X. We introduce an extra level of indirection to isolate the class interface from this implementation dependency. This approach is similar to the requester and implementor separation in CommonView[3], but in our case the interface class directly implements the external protocol.



---

```
class WindowRep;
public:
    // public functions
private:
    WindowRep* rep_;
};
```

Figure 5: Window class interface

---

For example, the definition of the InterViews window class is shown in Figure 5. The WindowRep class definition depends on X data types, which are not visible to the Window class user. WindowRep's data members are public, which violates one of our principles for library classes. However, the WindowRep class is not visible to the library user and the exposure of WindowRep simplifies the implementation of other X-dependent classes. Of course, we would prefer a more abstract interface but library internals are similar to an application in that the scope of a change is limited.

## 5 Documentation

The same goal for code that uses a class library holds for documentation: changes in the library implementation should not require changes to the documentation. This approach means that class documentation is a *subset* of the public and protected members defined in a class interface. In particular, functions that are present only for implementation reasons should not be documented.

The most obvious example of a public function present solely for implementation is a destructor, which is typically defined or not defined depending on whether an object contains pointers to other objects that might need to be deallocated. More subtle examples can occur for virtual functions that are defined or inherited depending on the implementation of the class. For example, the InterViews base class glyph defines a virtual function *undraw* to notify an object that its window area has been allocated to another object. A class will define undraw if information associated with its window position is cached. Thus, adding or removing caching affects whether the undraw function is present in the class' public interface, yet does not change the semantics of the object.

## 6 Conclusions

An encapsulated library of classes is not the same as a library of encapsulated classes. Through examples from our experience with InterViews, we have shown the benefits of hiding the implementation of object creation using a layer between the library user and a potential constructor call. Individual library classes have a greater need for protection than application classes. A library implementor should therefore make members private by default and treat a user's desire for a protection change as an enhancement request.

Implementation classes add a level of indirection to avoid pollution of a library user's namespace. In addition, this level of indirection eliminates the need to recompile the user code when the implementation class changes.

To avoid implementation dependencies, class library documentation is slightly different from the C++ public and protected interface. Functions that are present solely because of the implementation, such as related to memory management or caching, are not present in the user-level library documentation.

## References

- [1] G. Aitken. OI: A Model Extensible C++ Toolkit for the X Window System. *Proceedings of the 4th Annual X Technical Conference*, Boston, Massachusetts, January 1990.
- [2] P. Calder and M. Linton. Glyphs: Flyweight objects for user interfaces. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, October 1990, pp. 92-101.
- [3] F. Dearle. Designing Portable Application Frameworks for C++. *Proceedings of the Second USENIX C++ Conference*, San Francisco, California, April 1990, pp. 51-61.
- [4] D. Knuth. *The TeX Book*. Addison-Wesley, Reading, Massachusetts, . 1984.
- [5] M. Linton. The Evolution of Dbx. *Proceedings of the Summer USENIX Conference*, Anaheim, California, June 1990, pp.211-220.

# Sniff—A Pragmatic Approach to a C++ Programming Environment

Walter R. Bischofberger  
*UBILAB (UBS Informatics Laboratory)*  
*Bahnhofstrasse 45*  
*CH-8021 Zurich/Switzerland*  
*Phone: (0041) 01 236 31 83 (direct)*  
*Fax: (0041) 01 236 46 71 (direct)*  
*Email: bischi@ZH010.ubs.ubs.arcom.ch*

## Abstract

Sniff is a C++ programming environment which runs on different UNIX workstations under OSF Motif, OpenWindows, and SunView. Sniff is an open environment providing browsing, cross-referencing, design visualization, documentation, and editing support. It delegates compilation and debugging to any C++ compiler and debugger of choice. Sniff has been in internal use at UBS (Union Bank of Switzerland) since August 1991. Since then several developers are applying Sniff in writing serious software systems as well as in evolving Sniff.

In developing Sniff we took a pragmatic approach. We chose simple and efficient techniques in implementing Sniff's components and concentrated on combining these components into a seamless application. The aim of this paper is to describe Sniff's components and how they cooperate, to show the decisions which had to be taken, the trade-offs which have resulted, and to discuss our experience in applying Sniff.

## 1. Introduction

The main goal in developing Sniff was to create an efficient and portable C++ programming environment which makes it possible to edit and browse large software systems textually and graphically with a high degree of comfort, without wasting huge amounts of RAM or slowing down in an annoying way.

To achieve these goals in a reasonable time we decided to take a pragmatic approach. We chose simple and efficient techniques in implementing Sniff's components and concentrated on combining these components into a seamless application.

The aim of this paper is to describe Sniff's components and how they cooperate, to show the decisions which had to be taken, the trade-offs which have resulted, and to discuss our experience in applying Sniff.

Section 2 gives a short overview of Sniff's basic building blocks. Section 3 consists of a discussion of Sniff's overall architecture with emphasis laid on the topic of openness. The following three sections give in-depth information about Sniff's basic building blocks. Section 7 discusses the portability related aspects of Sniff. Section 8 compares certain aspects of Sniff with the solutions chosen in the development of other C++ development environments. The paper ends with a discussion of the experience gained in developing and applying Sniff.

In this paper we do not discuss the look and feel of Sniff's user interface. For this reason we included Figure 1, a screen dump which gives an impression of a subset of the tools Sniff provides. It was optimized for showing as many tools as possible and shows no typical working configuration

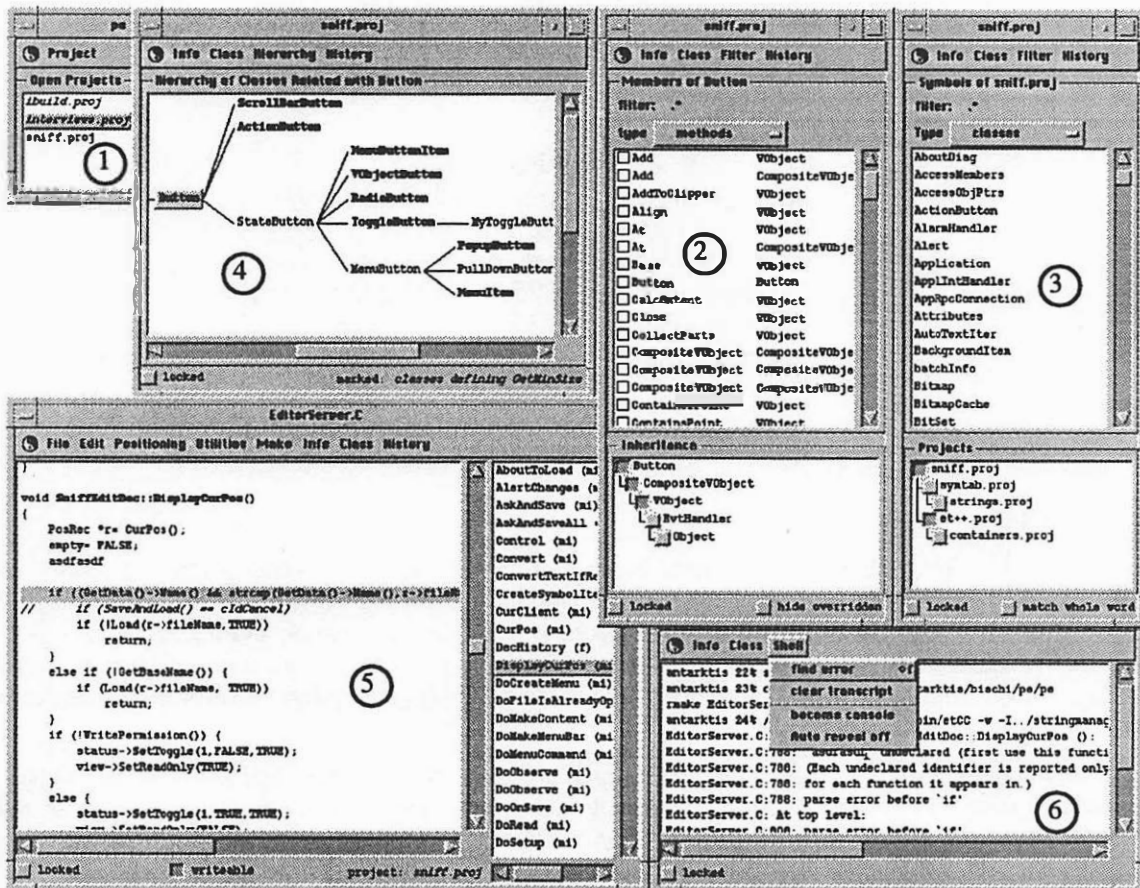


Figure 1. Overview of Sniff's user interface

## 2. The Building Blocks of Sniff

From a general point of view the tasks of a programming environment are to

- gather information about the software system under development
- update this information after the source code was modified
- give support for browsing and editing this information
- execute the software system
- provide means for inspecting the execution process (debugging)

To provide these services a complete programming environment has to consist of at least the following parts:

- an information extractor, which extracts information from the source code
- an information repository, which manages the extracted information, as well as certain project specific information
- an execution component, which compiles and executes, or interprets the software system and provides debugging support

- a user interface consisting of several tools which make it possible to browse and manipulate the information in the data repository as well as to carry out other programming environment specific tasks (e.g. generation of makefiles and software metrics)

Sniff's information extractor is a fast fuzzy C++ parser which runs as a server process, parses entire source files on request, and sends a tokenized stream of extracted information to the client. It can be accessed by any program which needs information about the internals of a C++ source file.

Sniff's information repository is a symbol table which is kept in main memory and which is rebuilt every time a project is loaded. After a source file was changed all information about this file is discarded and rebuilt. The symbol table stores information about where and how the symbols (e.g., classes, members, variables) of a software system are declared and defined. Information about where the symbols are used is extracted on request.

Sniff does not provide an execution and debugging component on its own for reasons of portability, efficiency, and manpower. It delegates compilation to the compiler of choice and debugging is carried out with the corresponding debugger. We use Sniff, for example, on our SUN workstations together with AT&T, SUN, and GNU C++ compilers and the corresponding debuggers.

Sniff provides several tools for browsing and manipulating a software system based on the information stored in the symbol table and on information which is collected on the fly. These tools provide support for

- project management (1)
- browsing the symbols declared in a software system (3)
- getting information about usage of symbols (Figure 4)
- accessing their source code (5)
- for browsing the interface of a class (e.g., which members are declared in the class itself and which members are inherited from which base class) (2)
- visualizing the inheritance graph and information about protocols (4)
- compilation management (6)
- displaying documentation about symbols
- editing and browsing source code (5)

The numbers behind the points reference the markings in Figure 1. They link the described functionality with the corresponding user interface elements on the screen dump (only a subset of the user interface element is displayed).

### 3. Openness and Tool Cooperation

One of the main topics in actual discussions about programming environments is openness and the closely related topic of how the tools of a programming environment communicate and cooperate.

Openness means in this context that the tools of a programming environment are decoupled in a way which makes it possible to replace existing tools or insert new tools without having access to the source code of the entire system and without having to mess around in a large monolithic source code.

Many authors such as Gabriel et al. [Gab90] and Reiss [Rei90a], [Rei90b] are arguing for absolute openness. This absolute openness is achieved by running all tools in separate operating system processes and by making the tools communicate via well defined protocols.

In the field of open programming environments the coordination of the cooperation of various tools (control integration) is not the only challenge. The other challenge is the sharing of the large amount of information which has to be handled by a programming environment (data integration). This is especially

true for modern programming environments because they provide sets of tools which visualize and manipulate the same information in different ways.

In developing a completely open programming environment we have to build tools which manage control and data integration. To support control integration we need a tool that manages cooperation and provides services such as message passing, starting of tools if they are not already running, and sensible handling of exceptions such as the breakdown of a certain tool. To achieve data integration we need a central data management unit which administers shared information and which makes this information available to arbitrary tools via abstract protocols. This central data management unit has to coordinate the updating of the information and to prevent consistency violating simultaneous updates. An example for such an open environment is Energize (formerly called Cadillac [Gab90]).

Openness and the resulting decentralization, achieved by communication over general common protocols, are general desirable goals in developing large software systems. Nonetheless, there are two reasons why their importance has to be relativated in developing the kernel of a programming environment.

First, the kernel of a programming environment consist of a closely set of cooperating tools sharing large amounts of information, and providing a homogeneous user interface. The advantage of a kernel with a completely open architecture is the possibility to replace entire tools if the protocols are known. This kind of replacement does not make much sense because there are no general purpose tools which could replace kernel tools. The text editor of a modern programming environment, for example, is usually very much customized providing not only editing but also browsing capabilities (examples are the editors of Energize [Gab90] and Sniff). Therefore, it would not make sense for a developer to integrate his preferred editor into such a programming environment.

Second, complete openness is expensive both in terms of development effort and runtime overhead. The runtime overhead is neglectable for many software systems because they do not need all of the available processing power. This is not the case for available C++ programming environments, as many C++ programmers painfully learned.

While we do not consider openness to be important in integrating the kernel of a programming environment an open architecture makes sense for flexibly integrating and replacing new and peripheral tools which do not need the same degree of data, control, and user interface integration as the kernel tools.

In designing Sniff we decided, therefore, for a medium degree of openness. We built a centralized core which contains the symbol table and all tools which make intensive use of the information stored therein. The class browser and the symbol browser are typical examples for kernel tools. We decentralized services which do not require the same degree of integration as kernel tools and which can be accessed via simple protocols (e.g., the information extractor and the compiler). Furthermore, we provided the external access manager which permits external tools to access the centrally managed information, as well as to send requests and queries to the kernel tools. The overall architecture of Sniff is depicted in Figure 2.

Figure 2 visualizes the operating system processes of which a running version of Sniff consists as well as the kinds of communication between them. The kernel process is depicted in detail showing its logical subsystems. It consists of the project manager and the symbol table which are accessed by all subsystems and which manage control and data integration. The subsystems of the kernel communicate with member functions calls. The peripheral tools are integrated with the kernel in the following ways:

- The information extractor communicates with the symbol table over a stream connection.
- The compiler is invoked via a command line interface tool.
- All other tools communicate with the kernel's control and data integration managers via the external access manager. The external access manager provides an open interface based on ET++'s general inter application communication mechanism and translates external requests into internal member function calls.
- The debugger is not connected directly. It is wrapped with ET++'s generic debugger front-end which provides a comfortable graphical user interface and handles communication with the kernel via the

external access manager. The debugger front-end uses the debugger's command line interface for communication.

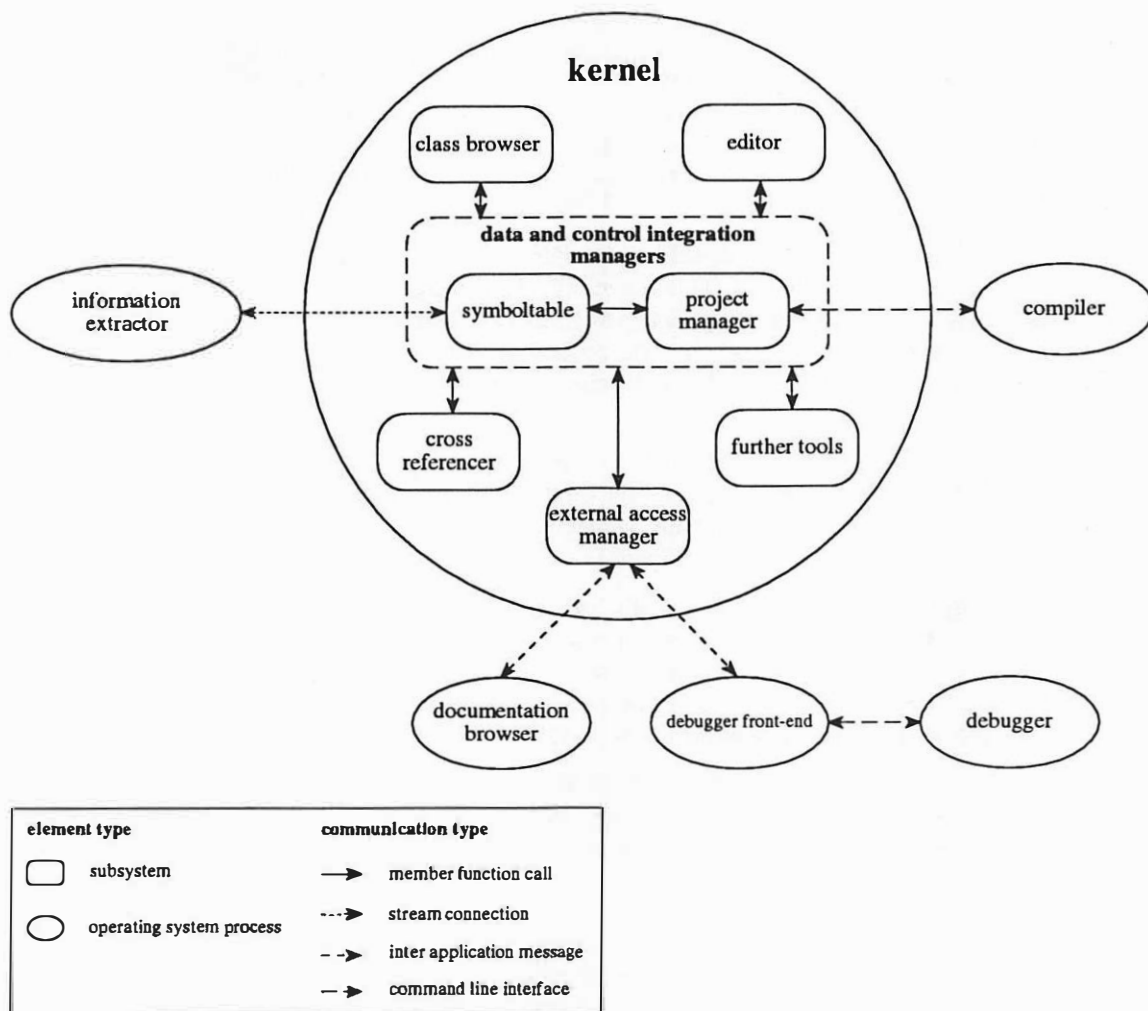


Figure 2: Overview of Sniff's architecture.

This design was chosen pragmatically to obtain a solution making use of the increased flexibility and openness provided by decentralization without having to pay the penalty in efficiency and in implementation costs resulting from complete openness. The centralization of Sniff makes it possible to extract the entire information about declaration and definition of symbols from the source code when a project is opened. Furthermore, it allows us to store the extracted information in a simple symbol table. Our approach eliminates possible performance and portability problems resulting from the use of a database shared by several processes and it eliminates all consistency problems rising from possible changes to source files between Sniff sessions. Furthermore we have no coordination and locking problems because the core tools run in one process and can therefore not attempt to change shared information in parallel.

A further advantage of our solution is that it becomes possible to update the information displayed in the open kernel tools without having to rely on complex incremental parsing and updating algorithms (which is a basic requirement in a completely open environment). After changes were applied to a source file Sniff reparses the entire file, updates the symbol table, and notifies all tools via change propagation. This easily implementable and maintainable scheme results in a completely satisfying runtime efficiency. Externally connected tools are notified about changes with a broadcast message.



The decentralized information extractor uses a simple stream based protocol to communicate with the kernel. It obtains requests for the parsing of a source file and it returns a tokenized stream of symbol related information which is analyzed by the symbol table. The decentralization of the information extractor has several positive effects. It makes it possible to share an information extractor between different instances of Sniff running on various workstations. It can positively affect response time if it runs on a fast server machine. It makes it easy to replace the information extractor which could, for example, make sense to give a developer the possibility to choose between a fast version without macro expansion and a slower version expanding macros (see also Section 4). A further interesting effect of the decentralization of the information extractor is that it makes it easy to support other object-oriented programming languages than C++, by simply writing an information extractor supporting the same protocol as Sniff's C++ parser.

Compilation is triggered from a command line interface tool by invoking the make facility or by calling any shell script provided by the user. This decentralization strongly increases portability and flexibility. Environments providing their own compilers do not only have to adapt them to changes in C++ (which are not scarce) but they also have to adapt code generation when the tool is ported to a new hardware architecture. A further advantage is that a developer can always work with the "best" available C++ compiler.

The connection of different debuggers is no trivial enterprise. Our approach is to provide a generic debugger front-end [Gam92] which can be instantiated for different kinds of debuggers. We currently work with a stand alone instantiations for the gdb and dbx debuggers. The next step in evolving Sniff is to integrate this generic front-end with Sniff as depicted in Figure 1.

In designing Sniff we have tried to pragmatically balance the advantages of openness with the development and runtime costs it causes. The result is a fast, user friendly, partially open programming environment. The degree of openness is appropriate for the current configuration of tools which were added with custom designed mechanisms. If our general access mechanism is appropriate will be evaluated when we integrate further tools such as the generic debugger front-end and design tools.

One architectural improvement we foresee is the unification of the external and internal control integration mechanisms. If the external interface for sending queries and requests proves suitable we will route the internal requests and queries over the external access manager.

#### **4. The Fuzzy C++ Parser**

The C++ parser is the foundation of Sniff because it provides most of the information which is managed by Sniff. The requirements for the parser were speed, adaptability and portability.

To satisfy these requirements we have developed a fuzzy recursive descent parser which has only a partial understanding of C++, which can deal with incomplete software systems containing errors, and extracts information about where and how the symbols (e.g., classes, members, variables) of a software system are declared and where they are defined. This way only declarations and the headers of (member) functions (i.e., return code, name and argument list) have to be parsed while the executable code in the bodies of (member) functions can be ignored. The extracted information is transmitted as a tokenized stream to the client.

The parser does not expand macros but parses the macro definitions. This makes it possible to drastically reduce the amount of code which has to be parsed (because included header files are only parsed once) and to provide information about which macros are defined where. The disadvantage of this approach is that macros can confuse the parser. (Non syntactical macros result in an error recovery process. Information about the next declaration may be lost. Macros used in software systems to be developed with Sniff should therefore be syntactical.)

Our approach makes only sense if the information loss and the extraction of erroneous information are negligible. For this reason we discuss some important applications of macros in C and C++ and the effect of not expanding them.

- 1 Macros are used as constants. These constants are usually used in the executable code and represent therefore no problem for the parser.



- 2 Macros are used to facilitate library integration by putting a unique, distinguishing string before all class names (e.g., "iv\_" in InterViews and "ET\_" in ET++). For a developer it is an advantage if the effect of these macros is not visible in the programming environments.
- 3 Macros are used to generate supporting code fragments from a few parameters such as class descriptors (in ET++ [Gam90]). The generated code is usually of no interest to the user and should be ignored in browsing. If these macros resemble declarations they can cause extraction of erroneous information.
- 4 Macros are used to hide differences between C++ versions. These macros are not syntactic if they are used to delete or replace key words. A typical example is the removing of the "= 0" for C++ versions which do not support pure virtual functions.
- 5 Macros are used to localize multiply used strings for improving changeability. E.g., the replacement of the name of the base class with a "SUPER" macro in the class declaration and in all places where an overridden member function is called. This application of macros leads to a heavy loss of information. Software systems containing this kind of macros should therefore not be developed with Sniff.

To prevent the problems caused by macros described in points 3 and 4 a developer has the possibility to define a list of strings to be ignored and a list of strings to be replaced with other strings. This simple mechanism makes it possible to ignore macros looking like declarations and to insert the correct key words where a macro was used to hide differences between different C++ versions.

During the last months Sniff was used to browse a lot of software systems and most of them did not use macros in a problematic way. Examples for such software systems are InterViews [Lin87] and ET++ [Wei89]. The only unsuited software system we tested is the NIH library [Gor87], where base class names are inserted with the macro "SUPER". This makes Sniff believe that the NIH library has a pretty shallow inheritance hierarchy. The problem of localizing the name of the base class could be solved elegantly by inserting a typedef for "SUPER" into each class declaration and by referencing the type "SUPER" to invoke overridden member functions [Str92].

In designing the C++ parser we had to balance parsing speed, adaptability, development effort, and the possibility to work with incomplete software systems containing errors against absolute correctness. We decided for a pragmatic solution. From our point of view it is more important to have a simple fast parser which works good for most software systems than to have a considerably slower parser which always extracts the correct information.

Most measurements we present were obtained based on the source code of the main ET++ directory. For this reason we list the most important information about the size and content of our test data in Table 1. If it is not mentioned explicitly the measurements were taken on a SUN SPARCstation1.

number of files	240
total size of files	860 KB
lines of code	XXX
number of classes	230
number of member functions	XXX 4000

Table 1. Properties of the source code used for measurements.

The actual source code size of the parser is 30 KB on 1500 lines of code. It was implemented and tested together with the symbol table in about three months. It parses the source files in the main ET++ directory (see Table 1) without constructing the symbol table in about 10 seconds. Together with the symbol table construction it takes about 20 seconds.

The adaptability of Sniff's parser is best exemplified by two extensions we applied. Once we recognized the importance of supporting old style C as defined in [Ker88] the parser was extended in one hour to correctly read the old parameter lists. Support for the template construct was implemented in two days (including the adaptation of the symbol table and the user interface).

With the growing speed of processors it could be possible that our pragmatic decision has to be revised. In this case we will be able to profit from Sniff's openness by writing a new parser which expands macros, has a better understanding of C++, and supports the same stream protocol as the old version.

## 5. Symbol Table and Cross Reference Information

A C++ symbol table is a complex data structure which has to be able to cope with large amounts of data. Sniff's symbol table is a nested structure of container objects provided by the ET++ foundation classes and basic information elements. The symbol table makes extensive use of hashed data structures. The class hierarchy of objects managed by Sniff's symbol table is visualized in Figure 3 which shows Sniff's hierarchy browser. The browser is displaying Sniff's class hierarchy restricted to all classes which are base or derived classes of class `SymtabObj`.

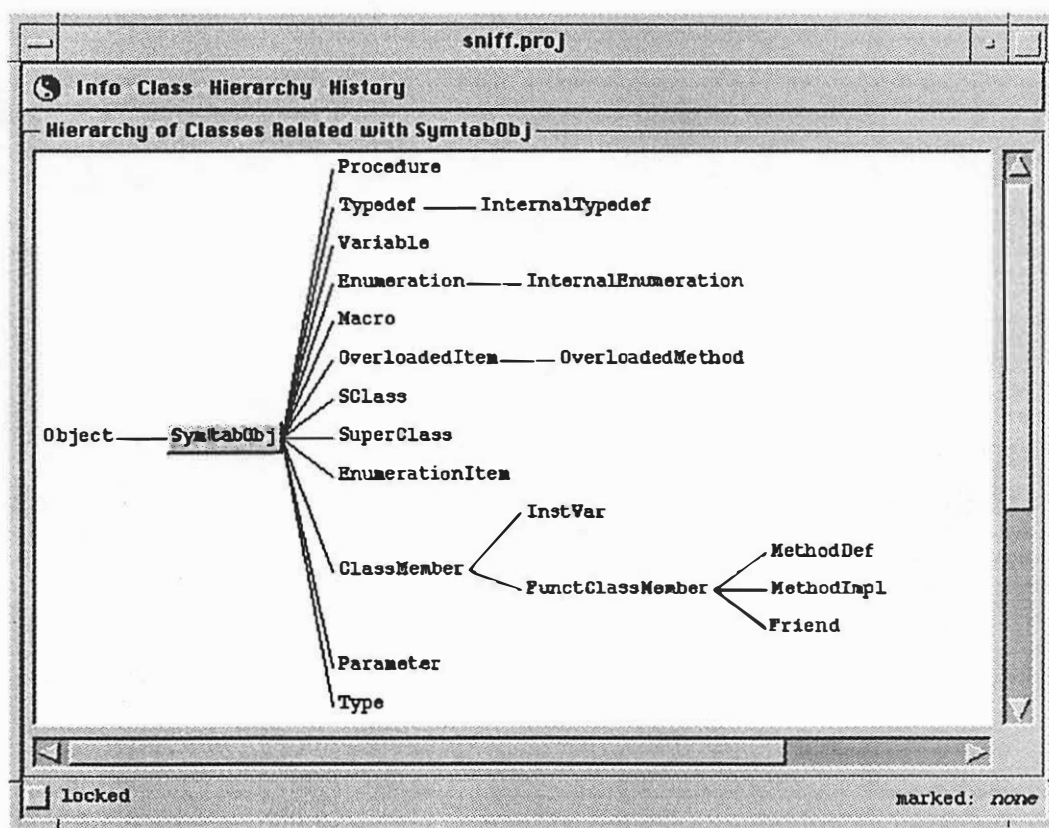


Figure 3: Class hierarchy of the objects stored in a symbol table.

The quality of a symbol table depends on what kind of information it stores, on the amount of storage it needs to represent a software system, and on the time required to retrieve an information subset. A symbol table storing the information extracted from the main ET++ source directory (see Table 1) needs about 3 MB of RAM. The time required to search the symbol table for all symbols matching a regular expression is less than a second.

cross reference information on the fly. The first prototype of the retriever tool shown in Figure 4 searched all source files sequentially with a regular expression matcher. The results were so satisfying that we did not consider any more sophisticated approaches for implementation.

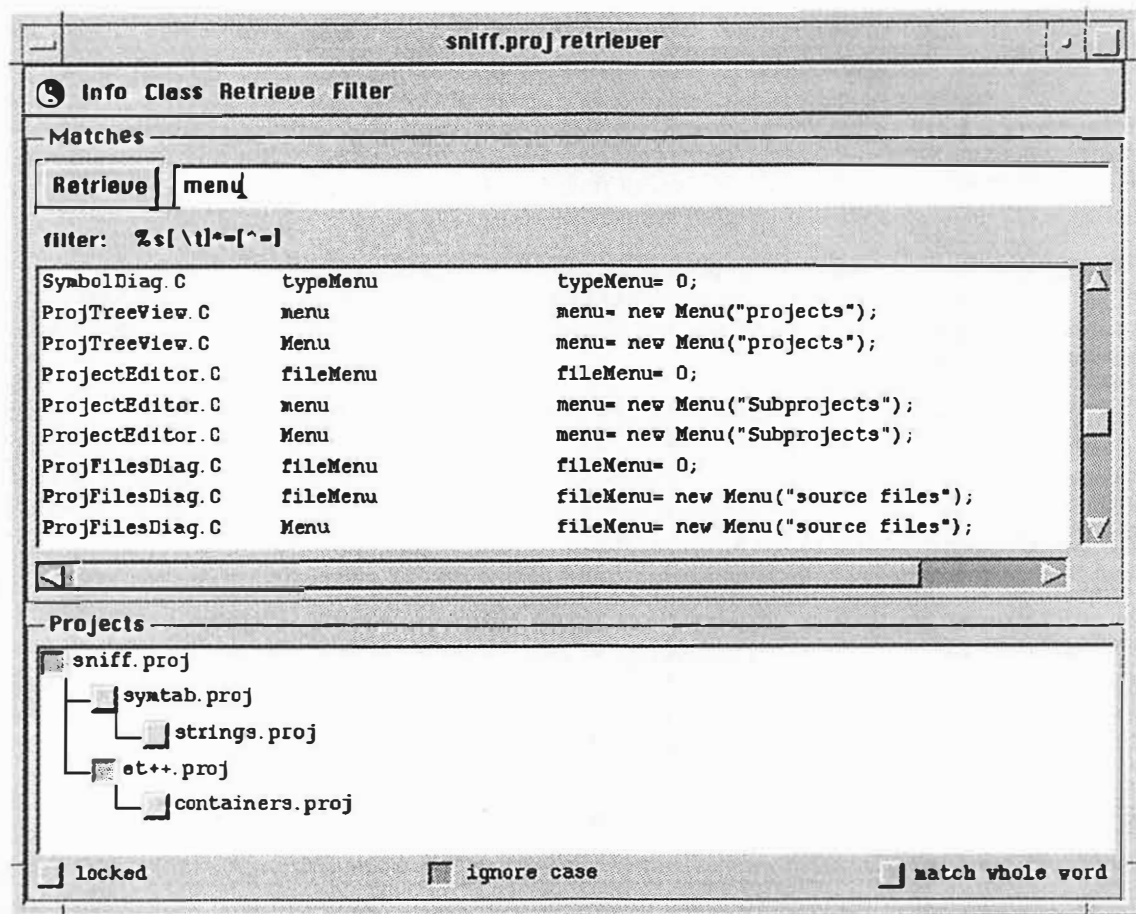


Figure 4: Sniff's retriever tool.

Sniff supports a similar approach as taken by developers searching their source files with the `grep` utility for occurrences of a regular expression. The difference is that Sniff provides considerable support for filtering and browsing the results of a query. The filter mechanism we use most frequently allows to restrict the results of a query to matches in the source files of a subset of the nested projects (see "Projects" subview in Figure 4). The following list describes a subset of further filtering criteria.

- Does the match consist of a whole word or substrings?
- Is the matching carried out case sensitive or not?
- Is the match part of an assignment or comparison operation?
- Is the match a parameter for an invocation of "new"?
- Is the match followed by a "(" and therefore probably a (member) function invocation?
- Does the line containing the match match a further regular expression?

The disadvantage of our approach is that a subset of the reported matches may be irrelevant. Fortunately, our experience in browsing large software systems has shown that this is almost never a problem because the relevant matches are easily recognized by a developer.

Experience has even shown that our approach is frequently more convenient than a conventional cross referencing mechanism which exactly reports all uses of a certain symbol. In using Sniff's retriever tool it is possible to issue fuzzy queries. These fuzzy queries make it possible to get information which is not easily detectable with classic cross referencing mechanisms.

Derived classes are, for example, frequently named by appending something to the name of the base class (e.g., Button, ActionButton, ToggleButton). By searching all occurrences of "Button" and applying the "new" filter all places can be obtained, where any kind of button is created on the heap. Another strategy we frequently apply bases upon the fact that in most software systems variables storing objects have sensible names containing parts of the names of the class of the objects they store. By retrieving all occurrences of the string "menu" and applying the assignment filter it is, for example, possible to obtain all places in a software system where a menu is assigned to an (instance) variable (see also Figure 4).

Fuzzy queries are very helpful in browsing large software systems because they make it possible to extract and browse related information in one step. Their usefulness depends only to a small degree on the search process itself but rather on the expressiveness of identifier names and on the cleverness of the employed filtering techniques.

Searching all strings matching a regular expression in 850 KB of ET++'s source code takes about 6 seconds on a SPARCstation1. In working with Sniff the range of projects which has to be searched for a query can easily be restricted so that according to our experience the average response time of cross reference queries is about two seconds. The symbol table lookup of all declarations matching the same regular expression takes less than one second as discussed in the previous section.

The implementation of Sniff's cross referencing mechanism is another example for our pragmatic approach. We had to balance the advantage of a slow space consuming solution providing *exact* results against the fast solution which can result in too many matches. We first decided for the fast solution because we go with Andrew Koenig in believing that "Interestingly, sometimes a fast answer that's slightly wrong, but almost right, is preferable to a slow one that's closer to being right." [Koe92]. In retrospect we believe that our solution is not only faster but also better because of the power of fuzzy queries.

## 6. Realization of the User Interface

Programming environments are highly interactive tools which visualize information about a software system and give support to manipulate and execute it. This is why their user interfaces consume a large percentage of the overall development effort and why they are an important factor affecting user friendliness.

While a discussion of the look and feel of Sniff's user interface goes beyond the scope of this paper we feel that it is important to discuss some technical aspects such as its overall architecture, the tools we used for its implementation, and the experience we made with both of them.

Sniff's architecture is characterized by the clear division between data management and user interface. The data management consists of the project manager which manages the project related information and the symbol table which manages information that is extracted from the source code (i.e., the model). The user interface consists of several tools which serve as views on project and symbol data. This model/view view of Sniff's architecture is depicted in Figure 5.

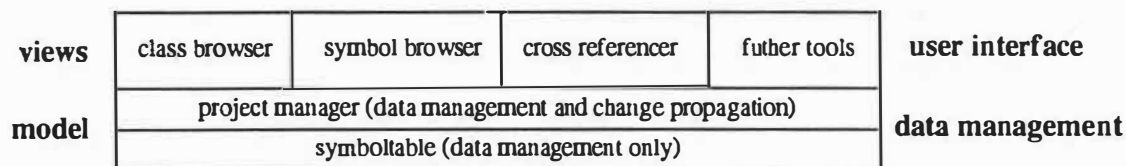


Figure 4. Sniff's architecture (model/view view).

The views extract and store data by using the access member functions of the project manager and the symbol table. From this point of view the project manager and symbol table would form one single data management layer. This is not the case because the project manager shields the symbol table in two ways. First, it is the only data managing part known by the views and it forwards symbol specific queries to the symbol table. Second, it is the manager which notifies the views about changes in the model via change propagation. The effect of shielding the symbol table is that the symbol table knows absolutely nothing about the views displaying its data and serves as mere data repository. The symbol table can therefore be reused in any other software system requiring information about C++ source code without having to be modified. The project manager is less decoupled from the user interface because it does not only provide information but has to use the same change propagation mechanism as the user interface tools.

Sniff's user interface was implemented with ET++ ([Wei88], [Wei89]), an object-oriented application framework. In rating the influence of ET++ on the development of Sniff we see two positive influences.

First, the reusable high-level parts of ET++ such as the graph browser, the text building block, the shell view, and the list manager strongly reduced the implementation effort of the user interface. It took us, for example, one day to implement the first version of the hierarchy browser, the tool for visualizing the inheritance graph and providing support to visualize queries such as "mark all classes related with class XXX and defining member function YYY".

Second, the standardized, reusable architecture embodied in ET++ made it possible to apply a prototyping-oriented, exploratory development strategy. Because we had an architectural framework it was possible to evolve Sniff instead of having to rewrite major parts after major redesigns of the user interface. According to our experience this is impossible if a toolkit providing only basic building blocks is used because in this case basic architectural decisions taken by the developer are frequently revised during redesigns. The most impressive experience we made in evolving Sniff was that it was possible to painlessly evolve the symbol browser, the first tool we had implemented as a quick shot to test the contents of the symbol table.

We did not only select pragmatic technologies in realizing Sniff but we took also a pragmatic development approach in starting with a small prototype which was then evolved into the first distributed version of Sniff. Our prototyping-oriented evolutionary strategy made it possible to evaluate the quality of the selected technical solutions early in the development process. Furthermore, it lead to a constant evolution of the user interface because we had a lot of feedback from early adopters. While the evolution of the user interface took us a certain amount of time it was carried out without painful global redesigns because of the high degree of reusability of the ET++ application framework.

## 7. Portability

An important goal in designing Sniff was to ensure portability, i.e., to make it possible to port Sniff to a new platform by simply recompiling it. While we did not yet practically validate the portability we are optimistic because of the following three reasons. Hopefully well have ported Sniff to at least one other platform when you read this paper.

- ET++, the application framework used to implement Sniff's user interface, is easily portable. It runs under OSF Motif, OpenWindows, and SunView on a number of different UNIX workstations.
- The problem of porting the execution mechanism, i.e., a compiler or an interpreter, does not occur in porting Sniff, because Sniff delegates compilation to any specified compiler running on the new platform.
- Sniff is implemented entirely in C++ and compiles on a wide range of C++ compilers such as AT&T cfront, SUN C++, and Gnu C++.
- The gdb instantiation of Etdbg, the generic debugger front-end, runs on many platforms.

## 8. Comparison with Similar Tools

In comparing programming environments it is possible to have a look at their external behavior (i.e., what kind of functionality do they provide with which response times) or it is possible to study their architecture and the approaches which were taken in implementing their components.

The external behavior can only be fairly judged if all tools to be compared were applied for a certain time in a real world development project. This is exactly what we plan to do once distribution of Sniff has started.

In this discussion we distinguish two kinds of tools. One kind are C++ browsers supporting the study of software systems which are not undergoing changes during browsing. The other kind are full featured programming environments which support browsing, editing, and execution of software systems.

Available or currently discussed C++ browsers are CIA++ [Gra90], the XREF tools [Lej90], and the SUN Source Code browser [SUN91]. The basic idea underlying these tools is to extract information with adapted compilers from C++ source code, to store this information in a data base and to provide tools to browse the data base. CIA++ and the XREF tools store their information in a relational data base which can also be directly accessed.

These tools differ from programming environments in that they do not have to ensure short update times. From our point of view such tools are interesting if they make it possible to directly query the data base. This is, because such a query facility makes it possible to obtain additional information which can not be obtained from the available browsing tools. For examples see [Gra92]. Direct data base access is not provided by currently available C++ programming environments.

In the rest of this section we compare some aspects of Energize from Lucid, ObjectCenter from CenterLine, and Objectworks\C++ from ParcPlace Systems—three commercially available C++ programming environments—with Sniff. We will set Sniff in perspective with Objectworks\C++ and ObjectCenter based on a bench mark study obtained from ParcPlace Systems [Par91]. All performance measurements stem from this study which was written by the developers of Objectworks. Its objectivity, therefore, has to be relativated. The time measurements were taken on a SPARCstation2.

We do not intend to make any statements about the quality of ObjectCenter or Objectworks\C++ because that would require extensive discussion about how the time measurements were obtained. We only want to show the order of magnitude of the differences with the measurements obtained with Sniff.

An important architectural difference between Energize, ObjectCenter, and Objectworks\C++ is their degree of openness. Energize has an open architecture with flexible mechanisms for control and data integration [Gab90]. ObjectCenter consists of two parts, a monolithic kernel and a decoupled user interface which makes it possible to integrate the kernel into other tools. ObjectCenter's kernel provides browsing, hybrid execution (interpretation and direct execution), and debugging. Objectworks\C++ has a monolithic architecture. Sniff's architecture lies in between by providing a centralized kernel with flexibly connected peripheral tools and by providing an interface for integrating other tools which do not need the same degree of control, data, and user interface integration as the kernel tools.

All three discussed tools extract their information about the source code either with an adapted compiler or with an interpreter or compiler in the case of ObjectCenter. The information is then stored in an object-oriented data base (Energize), in the object code (ObjectCenter compiler) or in files (Objectworks). The advantage of this approach is that the information extracted is certainly correct and that cross reference information can be extracted. The disadvantages of information extraction with compilers/interpreters is that it works only for correct C++ code (which may be problematic during extensive reorganizations). Moreover the information extraction is slower because of macro expansion and manifold parsing of header files.

According to ParcPlace's bench mark study the initial information extraction is time consuming. For extracting and loading the browsing information of the InterViews 3.0 ibuild software system consisting of about 36.000 lines of code (1MB disk space) ObjectCenter 1.02 needed 1 hour 21 minutes and Objectworks\C++ 2.4 needed 1 hour and 35 minutes. With Sniff the same activity took about 30 seconds.



Both ObjectCenter and Objectworks\C++ can store the information between sessions so that it does not take that long to open a project the second time.

ObjectCenter and Objectworks both keep all information about a loaded project in main storage, as Sniff does. The difference is that they store also cross reference information (and the interpretable representation of the source code in the case of interpretable files in ObjectCenter). This means that they need much more main storage than Sniff does. Energize keeps its information in an object-oriented data base. While this reduces main storage requirements it slows down accesses.

According to ParcPlace's bench mark study ObjectCenter 1.02 needs about 87 MB of main storage if all source files are loaded for interpretation Objectworks\C++ needs 19 MB of main storage. Sniff in comparison needs 5 MB with ibuild loaded and 7 MB when the InterViews library is also loaded. ObjectCenter and Objectworks\C++ would probably have problems to load InterViews together with ibuild.

There are two facts relativating these measurements. First, ObjectCenter would not be used this way. A developer would make use of the hybrid execution facility to obtain a sensible tradeoff between available information for browsing and main storage requirements. Second, both ObjectCenter and Objectworks\C++ have the information available to provide debugging support. The main storage requirements for Sniff grows significantly during debugging.

Sniff's approach has two advantages in the area of main storage requirements. First, the information required for debugging purposes is only loaded during debugging. Second, gdb [Sta89] makes it possible to load symbolic information on demand which reduces the overall storage requirement.

The updating of information after a source file was changed requires the recompilation (or preparation for interpretation) of this file in the case of ObjectCenter and Objectworks\C++, and an incremental compilation in the case of Energize. This takes certainly more time than the time it takes Sniff to update its information about a source file and update all open tools (1-2 seconds). If the change in the source code is tested immediately the approach of the commercial environments is no disadvantage, because Sniff has to compile the file too. Sniff's approach is an advantage if a developer makes several changes in different source and header files but wants to continue browsing.

## **9. Experience with Sniff**

Since we started using Sniff in August 1991 about ten persons have applied it for software development. During this time the C++ parser and the symbol table were almost unchanged while the user interface evolved considerably.

The main experience in developing Sniff is, from a technical point of view, that it is possible to write a user friendly, efficient, complete, portable C++ programming environment in one personyear by taking a pragmatic approach and by building on a powerful class library. The resulting software system consisting of 18,500 lines of C++ code is easily expandable and maintainable because of Sniff's partially open architecture and because our pragmatic approach resulted in the selection of simple and quickly implementable solutions.

Besides the technical point of view the users point of view has to be considered as well. While it is difficult to measure user friendliness everybody using Sniff feels that the way he browses and edits software systems has changed. This is especially obvious with novice ET++ programmers. According to our previous experience a novice ET++ programmer needed about two months to learn enough to become productive. The last two novices which used Sniff from the beginning became productive in less than half this time because Sniff made it much easier for them to understand the comprehensive application framework as well as the sample applications.

## Availability

Sniff can be obtained via anonymous ftp from [iamsun.unibe.ch](ftp://iamsun.unibe.ch) (130.92.64.10).

## Acknowledgments

Thanks to the referees for their constructive criticism. Thanks to Erich Gamma and André Weinand for their technical support and for many discussions about this paper. Thanks also to Andreas Birrer and Bruno Schäffer for many diverting lunches and the discussions we had about Sniff and this paper.

## References

- [Gab90] Gabriel RP et. al.: Foundation for a C++ Programming Environment. Proceedings of C++ at Work-90, Secaucus, New Jersey, September 90
- [Gam90] Gamma E, Weinand A: ET++—A Portable C++ Class Library for a Unix Environment. OOPSLA Tutorial, Ottawa, Canada, October 1990
- [Gam92] Gamma E: Etdbg—A Generic Debugger Front-End. UBILAB Technical Report, 1992
- [Gor87] Gorlen KE: An Object-Oriented Class Library for C++ Programs. *Software—Practice and Experience* Vol. 17., No. 12, 1987
- [Gra90] Grass JE, Chen YF: The C++ Information Abstractor. *USENIX C++ Conference Proceedings*, San Francisco, CA, April 1990
- [Gra92] Grass JE: Object-Oriented Design Archaeology with CIA++. *USENIX Computing Systems*, Vol.5, No.1, 1992
- [Ker88] Kernighan BW, Ritchie DM: *The C Programming Language* (Second Edition). Englewood Cliffs, New Jersey, 1988
- [Koe92] Koenig A: Posting to the `comp.lang.c++` news group on the USENET, 1992
- [Lin87] Linton MA, Calder PR: The Design and Implementation of InterViews. *USENIX Proceedings and Additional Papers C++ Workshop*, Santa Fe, NM, 1987
- [Lej90] Lejter M, Meyers S, Reiss SP: Adding Semantic Information to C++ Development Environments. *Proceedings of the C++ at Work Conference*, Secaucus, NJ, September 1990
- [PAR91] ParcPlace Systems: A Performance Comparison of Objectworks/C++ and Saber-C++ Development Environments. ParcPlace Systems, Sunnyvale, CA, 1991
- [Rei90a] Reiss SP: Interacting with the FIELD Environment; *Software—Practice and Experience*. Vol. 20, No.1, 1990
- [Rei90b] Reiss SP: Connection Tools Using Message Passing in the Field Environment. *IEEE Software*, July 1990
- [Sta89] Stallman RM: *GDB Manual—The GNU Source-Level Debugger*. Free Software Foundation, Inc., Cambridge, MA, 1989



- [Str92] Stroustrup B: How to write a C++ language extension proposal. C++ Report, Vol. 4, No. 4, May 1992
- [SUN91] Sun Microsystems: Sun SourceBrowser Reference. Sun Microsystems, Mountain View, CA, 1991
- [Wei88] Weinand A, Gamma E, Marty R: ET++—An Object Oriented Application Framework in C++. OOPSLA 88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, 1988
- [Wei89] Weinand A, Gamma E, Marty R: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. Structured Programming, Vol. 10, No. 2, 1989



# A Statically Typed Abstract Representation for C++ Programs

Robert B. Murray  
rbm@mozart.att.com

*AT&T Bell Laboratories  
600 Mountain Avenue  
Murray Hill, NJ 07974-2070*

## ABSTRACT

Alf (A Language Foundation) is a complete, statically typed, abstract representation for C++ programs. Alf represents the program semantics, not the syntax. It has been designed to be easy to analyze and modify, which allows tools to operate on C++ programs without having to be able to parse and typecheck C++. Important design goals included taking full advantage of static typing (to catch logic errors at compile time); abstraction (implementation details are private); ease of analysis and manipulation; and the possibility of a compact representation. This paper presents a high-level overview of Alf, including an example Alf-based analysis program.

## 1 Introduction

The development of new C++ tools is hindered by the fact that C++ source is difficult to parse and typecheck correctly. A tool that wants to do a conceptually simple task (such as read a set of class declarations and produce a picture showing the inheritance hierarchy) must first solve the much more complicated problem of deducing the inheritance relationships from the source. With the current technology, the tool builder has essentially two choices:

### 1.1 Do it right: a full understanding of the language

Since the cost of building a program that really understands C++ is high, most tools that do it right do so by taking an existing copy of the source for some C++ compiler and introducing changes. In addition to the costs of acquiring the source in the first place, the developer must understand the compiler well enough to introduce the necessary changes, and must be prepared to reintroduce those changes into subsequent releases of the compiler. This is prohibitively expensive for all but the most critical tools.

### 1.2 Fake it: a partial understanding of the language

An alternative is to write a quick and dirty parser that ignores the parts of the language that the tool doesn't care about. Such tools tend to be ignorant of issues involving scoping and the type system, and are usually single-purpose (they know just enough to do one task); this makes them hard to extend.

Since building a correct parser is much harder than building a parser that just gets the easy parts right, this approach also tends to build tools that produce results that are often, but not always, correct. For instance, a tool that looks for inheritance relationships might just look for a token sequence of the form:

```
class classname : tokenlist {
```

and treat every token in the *tokenlist* that was not a comma, `public`, `private`, or `protected` as the name of a base class. Even if we assume that such a tool is smart enough to ignore tokens in comments and quoted strings, it can still be fooled by code that uses qualified names or class templates:

```
class X : public Foo::Bar, public Temp<(sizeof(int) > 1<<4 )> {
```

To get this right, our tool must not only understand qualified names, but must be able to do at least enough expression parsing to figure out which `>` terminates the template argument. Even worse, if for some reason the tool wanted to examine the definition of `Foo::Bar`, it would have to perform C++ name resolution — which is a task that is not only hard for compilers to get right, but is subject to future decisions of the ANSI/ISO C++ standards committee. We may be willing to accept occasional errors from a tool that prints inheritance hierarchies; but in general, tools that are “often correct” will be of limited utility.

These problems can be solved by having a single program (a parser) translate C++ source into a representation that other tools can easily analyze and manipulate. The parser is smart enough to “do it right”. Other tools do not need to understand the details of the C++ syntax; they need only concern themselves with the semantics. In the same way, typechecking information (including name resolution) can be the responsibility of a single tool; that tool can record its decisions in the internal representation for other tools to refer to.

This paper describes one such internal representation (Alf) that represents C++ programs as trees of objects.

## 2 The Foundation Project

Alf has been developed as part of a larger effort to develop a framework (named Grail<sup>[10]</sup>) for programming environments and tools. Over the last two years, we have made four major iterations through the design of Alf; at each iteration, the design and implementation were refined based on feedback from a variety of sources, including performance measurements, the evolution of Grail, and the experiences of several friendly (but adventurous) developers who built Alf-based tools. We anticipate that the evolution of the design will continue.

## 3 Related Work

The idea of a semantically analyzable internal representation is not a new one. The Interlisp system<sup>[13]</sup> stored programs as objects, not as ASCII text; it had the benefit of representing

an underlying language whose structure is much simpler than that of C++. The same is true for the Mjølner BETA System<sup>[2]</sup> environment, which stores BETA programs as a collection of objects.

The MENTOR project<sup>[1]</sup> created a representation for Pascal; this work included a separate language to analyze and manipulate the program representation. Alf takes a different approach: the Alf abstraction is a set of C++ objects, not a new language.

DIANA<sup>[9]</sup> is an internal representation for Ada<sup>®</sup> programs; Alf is very much like a DIANA for C++. Both approaches represent the entire semantics of the target language in a regular form that is designed to be easy for other tools to analyze.

Alf expresses a C++ program as a collection of C++ objects; this allows the use of inheritance to encapsulate the many is-a relationships between the objects. Alf was designed to have a compact representation; this should avoid the problems of scale that large projects encountered when they used DIANA. DIANA includes a human-readable (ASCII) format; Alf does not. Alf also has no formal specification, as it reflects the semantics of a language (C++) that is itself not formally specified.

Reprise<sup>[3]</sup> is an adaptation of the IRIS internal representation<sup>[4]</sup> to C++. Alf is an abstraction, not a representation; it presents an abstract interface that could be implemented in a number of ways. Unlike Reprise, the Alf abstraction is statically typed (see section 6); this helps ensure that programs that violate the abstract model will cause compile time errors instead of run time errors. Alf is just one part of a larger effort (Grail) that has the goal of providing an infrastructure for a surrounding environment and tools. The current Alf implementation has been optimized for space compaction (see section 8).

The C++ Information Abtractor (CIA++)<sup>[6]</sup> does not store all of the information in a C++ program. This tool analyzes the relationships between C++ modules; it does not remember internal details (such as comments, or expressions not involving external references) that would not affect its analyses. We anticipate that building a tool to populate the CIA++ database from Alf trees should be straightforward, and would in fact be a good demonstration of the abstraction.

GENOA<sup>[8]</sup> is a language-independent code analyzer that can be interfaced to any system that represents programs as attributed parse trees. Although the trees in Alf represent the semantics, and not necessarily the parsing structure, GENOA ought to be well suited for writing inquiries on Alf trees.

The CIN++ interpreter<sup>[7]</sup> uses its own internal representation, based on the representation used by the CIN C interpreter; the CIN++ developers are studying the feasibility of porting their interpreter to Alf.

## 4 Semantic Structure

An Alf tree reflects the semantic structure of the program, not the syntactic structure. A separate typechecker analyzes unchecked Alf trees and records typechecking information in those trees; for example, every symbolic reference is linked to its corresponding declaration:

```
extern int i; // Declaration
```

```
main(){
    i;    // "i" linked to the above declaration
```

All other tools simply refer to the typechecking decisions recorded (by the typechecker) in the Alf.

For example:

```
int t,p;

class C {
    typedef int t;
    void f();
};

class D {
    void f();
};

void
C::f() {
    t * p; // #1
}

void
D::f(){
    t * p; // #2
};
```

Statement #1 defines an automatic object named p whose type is C::t\*; statement #2 multiplies the global integers t and p and discards the result. It would be hard for a syntax-based tool to reliably get this right; an Alf-based program has no such problem. The two statements have different semantics, so they are represented by objects of different classes; the fact that the two statements have similar syntax is of interest only to the parser.

Separating the parser and typechecker allows programs other than the parser to generate Alf trees which can then be passed to the typechecker.

## 4.1 Completeness

Alf trees preserve the meaning of the entire program (including comments). An *unparser* recreates a human-readable equivalent version of the program from the Alf trees. This means that a system using Alf doesn't have to preserve the original source: ASCII can be viewed as just one mechanism for creating the Alf objects that represent a program. When a human needs to make a change, the unparser can generate human-readable source from the old Alf; after editing this source, the parser and typechecker can generate the new Alf.

Of course, Alf-based systems may elect to keep the original source; that is a policy decision of the surrounding environment.

Note that the reconstituted ASCII source will be semantically equivalent to the original input source, but in general it will not be identical. This is because there may be more than one way to express the same semantics in C++. For example,

```
int a,b;
```

and

```
int a;  
int b;
```

are semantically identical in C++ and therefore generate the same Alf (two successive definitions of integers). Whether the unparser prints the first form or the second is a feature of the unparser, not of the Alf.

Storing a program as Alf objects makes it immediately available to any analysis or transformation tools. This approach also obviates arguments over the “right” way to display a program. The unparser is tunable; display decisions (e.g., where the braces go) are made by the person viewing a code fragment, not by the person who wrote it. One person looking at a program may choose to view it like this:

```
int  
main(int argc, char* argv[]){  
}
```

while another person may simultaneously view it like this:

```
int main(int argc, char *argv[])  
{  
}
```

Both people are looking at the same program; each has told the unparser to display it according to his or her own personal preference.

## 5 A note about the preprocessor

C programmers use the preprocessor for three main purposes:

- interface specifications (`#include`);
- macros (`#define`);
- conditional compilation (`#ifdef`).

Alf is intended to support these uses, either directly or by conversion to other constructs that do equivalent things. For instance,

```
#define BLOCKSIZE 1024
```

might be converted to:

```
const int BLOCKSIZE = 1024;
```

However, Alf cannot directly represent programs that engage in clever cpp tricks (such as redefining keywords). (Alf could be used to represent the cpp output; but reconstituting that output by running the unparser would not preserve the original cpp directives.) We anticipate that any Alf-based environment would have to support arbitrary C++ code, but that programs using ugly cpp tricks would not have access to the full range of features provided by the environment.

## 6 Static typing

There is a separate Alf class for each important concept in C++ (about 95 classes in total). These classes form an inheritance hierarchy, rooted at class `Alf`, that reflects the structure of C++. Operations are provided only at appropriate points in the hierarchy.

Because the structure of Alf reflects the structure of C++, the compiler can help check that programs using Alf do so in a way that is consistent with the language semantics. For instance, a program that attempts to ask an expression for its base classes will get a compile time error, because the `Alf Expr` object has no `base_classes` member function. (The toolsmith really meant to ask if expression's *type* was a class type, and if so, what the base classes were.) If the Alf abstraction did not reflect the structure of C++, toolsmiths would still have to understand that structure; but programs that violated that structure would result in run-time errors instead of compile-time errors.

The complexity of C++ makes this especially important. For instance, the Alf type system reflects the fact that extern-C linkage is part of a declaration, and not part of a function type. This makes it less likely that toolsmiths will forget about C++ constructs such as:

```
extern "COBOL" int i;
```

or even

```
extern "C" class Foo {  
    friend int bar(); // "bar" has C linkage  
};
```

For those cases where the exact type of an Alf object is not known at compile time (e.g., it must be an expression but we don't know what kind of expression), Alf provides a run time type identification system — see section 7.



## 7 Abstraction

Alf presents an abstract interface that completely hides the implementation. There is no public data. Alf client programs cannot ever directly touch the underlying Alf nodes; instead, all access is through handle objects (smart pointers). These handle classes are arranged in an inheritance hierarchy that reflects the *is-a* relationships between the underlying Alf nodes. This allows the Alf implementation to manage the memory for the underlying Alf nodes; it allows the use of a variety of representations for the Alf objects; and it allows the Alf objects to be physically distributed (for instance, project-wide or company-wide resources might be accessed across a network).

The Alf objects act like pointers, including the behavior that a handle to a base class can actually be “pointing” to a node that is of some derived class. For example, given that the class `If` is derived from class `Expr`:

```
If if1 = something();
Expr e = if1; //OK
If if2 = e;    //Static type error
```

Alf supports explicit run-time type identification <sup>1</sup> in the form of a safe downcast operation:

```
Cast<If> safe_cast = expr;
```

The actual type of the node pointed to by `expr` will be examined at runtime, and `safe_cast` will either point to the same node (if the downcast succeeds) or not point to any node at all (if it fails).

The definition of the `Cast` template is:

```
template <class T> class Cast : public T {
public:
    Cast(const Alf&);
};
```

A `Cast<T>` *is-a* `T` with a different constructor that does the run-time type identification. This means that a `Cast<T>` can be used as a `T` if the `Cast` succeeds:

```
Cast <If> an_if = expr;
if (an_if)
    do_something_with(an_if.then_part());
```

---

<sup>1</sup>This mechanism was designed before the current proposals for run-time type identification in C++ [15] were promulgated; if C++ is enhanced to provide safe run time type identification, we would change Alf to use that facility.

All runtime type inquiries are safe (they always build either a valid handle, or a handle that points to no node); and they must always be called explicitly.

The use of inheritance allows an Alf-based program to selectively ignore details of the objects it is examining. For instance, a program that is only interested in names that are declared can use a `Decl`, which is a handle that can point to any declaration; the inquiring program doesn't need to know the details of the declarations it is examining:

```
Alf a = something();
Cast<Decl> dcl(a);
if (dcl)
    cout << dcl.name().fullname() << " is declared.\n";
```

The `Cast` succeeds for any underlying node that is a declaration — including declarations or definitions of functions, data, classes, unions, typedefs, or enums.

All member functions on Alf's return a character string, an integer, an Alf, or an iterator for some Alf class. For instance, `Scope::statements()`, when called on a `Scope`, returns an `Alfiter<Stmt>`; this iterator can be used to generate a sequence of `Stmts` (statements). The use of an iterator allows the Alf abstraction to be independent of the data structures used by the Alf implementation.

## 8 Persistence

Alf objects are persistent; they can be stored in a database for later retrieval. The Alf abstraction hides the underlying database; we are currently using two different implementations of persistence, one that uses regular UNIX<sup>®</sup> files, and one that uses a commercial database. Steven Buroff has designed and implemented a persistence abstraction that frees both the Alf programmer and the Alf implementation from having to know about the underlying database. We anticipate that this will facilitate the porting of Alf to other database packages as they become available.

The current Alf implementation uses Andrew Koenig's *Shrubs*<sup>[5]</sup> to store the Alf trees as compactly as possible. While this is an implementation detail that is not part of the Alf abstraction, we anticipate that the compactness of the implementation will be instrumental in avoiding I/O bottlenecks.

## 9 Incremental Compilation

Alf is intended to support general analysis of C++ programs, but we are particularly interested in support for incremental compilation. All declaration/reference relationships in Alf are two-way — given a declaration, a program can find all of the named references to that declaration. This facility is necessary to allow an analysis program to determine which functions might be affected by a particular change.

## 10 Attributes

Alf cannot hope to anticipate all information that might need to be associated with Alf objects. Any program can associate an arbitrary object (known as an Attribute) with an Alf node; any program can retrieve any Attribute by supplying a key (which is an arbitrary character string). In this way, a user tool can remember certain things about an Alf object, without affecting other tools that may be using the same object (as long as the keys do not collide).

## 11 The Fundamental Alf classes

Class Alf (at the root of the Alf hierarchy) encapsulates functionality common to all the Alf classes. This includes operations to navigate through the Alf trees; run time type identification; persistence; debugging; and Attributes.

Almost all Alf objects are derived from at least one of the following classes:

**Scope** A Scope is a sequence of Stmt, where a Stmt (statement) is either an expression, a declaration, or a label.

**Expr** An Expr is an expression; the operands of an expression are its children. Alf expression trees are pretty much like any other expression trees.

**Decl** A Decl represents a declaration of something. Figure 1 shows the relevant portion of the Alf inheritance hierarchy; the classes in the lighter font are “abstract” in the sense that there can be no underlying nodes of those types, although there can be Alf handles of those types.

Declarations fall into two categories: types and nontypes. The nontypes are further divided into those that require a definition (functions and data, derived from `Defined_decl`), and those that do not (enumerators and nonstatic data members of classes). A `Defn` is the definition of a function or data. In a semantically correct program, every `Defined_decl` that is referenced by any `Name` (q.v.) will be *linked* to a single `Defn`. (Note that a `Defn` is a `Defined_decl` that is linked to itself.) Note also that a function definition (`Function`) is (using multiple inheritance) both a function declaration (`Function_decl`, which specifies the name and type of the function) and a definition (`Defn`, which contains backwards links to the declarations that link to this definition). In the same way, a data definition (`Data`) is both a data declaration (`Data_decl`) and a definition (`Defn`). Declarations of enumerations (`Enum` for the type, `Enumerator` for the individual enumerators) and nonstatic data members of classes (`Data_member`) have no corresponding definition, so these declarations are not derived from `Defined_decl`.

**Label** A Label represents a label, including case and default labels.

**Name** A Name is a symbolic reference to a declaration (`Decl`). Names can refer to types (`Type_name`) or nontypes (`Nontype_name`). For example, the `Defn` for this code:

```
String t;
```

will have as children the `Type_name String`, and the `Nontype_name t`. In a semantically correct program, every `Name` will be *resolved* (by the typechecker) to a declaration of that name.

**Type\_des** A `Type_des` (type descriptor) is a reference to a type. Classes derived from `Type_des` implement built in types, declared types, and undeclared types (such as pointers, references, and function types.)

## 12 Multiple Inheritance in Alf

Alf uses multiple inheritance when a particular Alf class belongs to the set of objects specified by the intersection of two base classes. As we have already seen, the base class `Function_decl` specifies objects that declare functions, and the abstract base class `Defn` specifies definitions. A function definition *is-a* declaration of that same function, and also *is-a* definition; multiple inheritance was the natural way to express this. As another example, a `Type_name`, which is a reference to a declared type, is both a `Name` (symbolic reference to a `Decl`) and a `Type_des` (reference to a type). Note that there are `Names` that are not `Type_des`'s (the `Nontype_names`, which refer to functions or data), and there are `Type_des`'s that are not `Names` (the descriptor for `char*` does not refer to anything that is declared).

Early in the design of Alf, we attempted to avoid multiple inheritance, since we believed that it would make the inheritance tree (and the program) more complicated. These attempts involved representing a single C++ object by two Alf objects (one for each "base class"); for instance, a function definition might have been represented by a definition object (which held the links to the declarations), and by a function declaration object (which held the name and type of the function).

This approach didn't work for two reasons. Representing something that is conceptually one object with two objects was awkward; there were many cases where the program got one half of the object and had to switch to the other half. Furthermore, there was usually extra information (e.g., the function body) that was only associated with the "derived class" part — not with either of the "base class" parts. In such cases, the single abstract object would have been represented by *three* objects. The use of MI avoided all of these problems, and in fact made the representation simpler, not more complex.

## 13 An example

Below is a simple function that, when passed the root of an Alf tree generates a Dag<sup>[11]</sup> input file that specifies a picture of the inheritance hierarchy of all of the classes defined in the tree:

```
1 #include    <iostream.h>
2 #include    "Decl.h"
3 #include    "Traversal.h"
4
```

```

5 const String quote("\\");
6
7 void
8 hier(const Alf& root)
9 {
10     Traversaliter iter(allpost.traverse(root));
11
12     cout << ".GS\\n";
13
14     Alf alf;
15     while(iter.next(alf)){
16         Cast<Class> clss(alf);
17         if(!clss)
18             continue;
19
20         String derived(clss.name().fullname());
21         Alfiter<Inherit> bases(clss.base_inherits());
22         Inherit inherit;
23         while(bases.next(inherit)){
24             String base(inherit.base_class().fullname());
25             cout << quote << derived << quote << " "
26                 << quote << base << quote;
27             if(inherit.is_virtual())
28                 cout << " label virtual ";
29             cout << "\\n";
30         }
31     }
32     cout << ".GE\\n";
33 }

```

The code on line 10 creates a `Traversaliter` object; this is an `Alfiter<Alf>` that generates, in the proper order, the nodes visited by a particular walk through an `Alf` tree. The `Traversal allpost` is a predefined object that specifies a bottom-up (postorder) traversal of the entire tree.

Each time the `Traversaliter` is poked by a call to `next` (line 15), it sets the value of `alf` to point to the next node in the traversal. Lines 16 – 18 examine this node, and if it is not the declaration of a class it is skipped. If it is a class, the call at line 21 returns an `Alfiter<Inherit>` that can be used to generate a sequence of `Inherit` objects; each `Inherit` specifies a base class, and also specifies whether that base class is public, private, protected, and/or virtual. The while loop starting at line 23 generates the `Dag` directive to draw an arc from the derived class to each base class.

Given the `Alf` objects for this example:

```

class Base {
public:
    class Nested {};

```

```
};
class D1 : public virtual Base {};
class D2 : public virtual Base {};
class D12: public D1, public D2, public Base::Nested {};
```

the `hier()` function produced this output:

```
.GS
"D1" "Base" label virtual ;
"D2" "Base" label virtual ;
"D12" "D1";
"D12" "D2";
"D12" "Base::Nested";
.GE
```

which, when run through Dag, generated the picture in figure 2.

## 14 Status

The current version of Alf supports the entire C++ language and is being used by several local tool developers. Peter Juhl has written a C++ parser that generates Alf; we have also implemented an unparser that translates Alf into C++ source. We are using the parser and unparser to test the system by feeding real programs to the parser, unparsing the resulting Alf, and compiling the output of the unparser with Cfront. The current implementation uses Koenig's Shrubs package and Buroff's persistence package to store Alf objects to disk.

Areas remaining to be implemented include full support for conditional compilation, typechecking, code generation, and templates.

## 15 Experiences

Static typing has proven to be useful in getting programs to run correctly. Many (but naturally not all) logic errors manifest themselves as type errors at compile time. We have found that the places where the structure of Alf was a surprise usually turned out to be places where there was a part of C++ that was not well understood by the toolsmith (e.g., the fact that a class can be `extern "C"`).

Choosing good names for a hierarchy of 95 classes is both important and difficult; it took several iterations to get it right. Many discussions that began as apparent disagreements about the correctness of an abstraction turned out to be simple misunderstandings caused by a confusing or imprecise class name.

Developers who have used Alf to build analysis tools have been pleased with the results: the development time is typically a small fraction of the estimated time to build the tools from scratch. (As an example, the `hier` function presented here was written and debugged in less than an hour; it replaces a hand-coded version that took a week to implement, and

had to be updated each time the C++ language definition changed.) Building tools that transform existing Alf (e.g., adding extra member functions) has been demonstrated but takes some skill and care. We are reminded of a comment by Gouge et. al in [1] :

Even the easiest and most natural program transformations are hard to implement in a totally safe way in the current state of baroque-ness of programming languages ... The more mundane transformations have proved to be challenging and interesting research problems.

The "baroque language" referred to in the quotation is Pascal. See also the discussion in [12] .

We anticipate that the development of a library of tools and libraries for program transformations using Alf will make things easier; but building 100% safe program transformations will never be trivial.

## 16 Availability

The Grail project, including Alf, is still under development; no products are currently available.

## 17 Summary

Alf presents C++ programs as trees of abstract objects. These objects are much easier to analyze and manipulate than the original C++ source; this, plus a surrounding structure, makes the development of tools that understand and manipulate C++ programs much more practical and affordable.

## 18 Acknowledgments

Peter Juhl wrote the parser; Andrew Koenig designed and implemented the Shrubs class that allows us to have a compact representation; Steven Buroff designed and implemented the underlying persistence package; and Bjarne Stroustrup designed the Grail architecture, as well as providing the original impetus for the Foundation Project.

These four also contributed important ideas and feedback, as did Martin Carroll, Stan Lippman, Barbara Moo, Bill Opdyke, and Judy Ward.

## References

- [1] Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B., Programming Environments Based on Structured Editors: The MENTOR Experience, in "Interactive Programming Environments", McGraw-Hill 1984.

- [2] Nørgaard, C., and Sandvad, E., "Reusability and Tailorability in the Mjølner BETA System", Proceedings, TOOLS '89, Technology of Object-Oriented Languages and Systems, Paris, November 1989.
- [3] Wolf, Alex, and Rosenblum, David; "An Overture to Reprise, A Representation for Semantically Analyzed C++ Code", Proceedings 1991 USENIX C++ conference.
- [4] Baker, D., Fisher, D., and Shultis, J., "The gardens of Iris", Technical report, Incremental Systems Corporation, Pittsburgh, PA 1988.
- [5] Koenig, Andrew, "Space-efficient trees in C++", Proceedings 1992 USENIX C++ Conference.
- [6] Grass, J., and Chen, Y., "The C++ information abstractor", Proceedings of the Second C++ Conference, USENIX, San Francisco, 1990.
- [7] Kowalski, T. et. al, "A Reflexive C Programming Environment", International Workshop on UNIX-Based Software Development Environments, Dallas Texas, January 1991
- [8] Devanbu, P., "GENOA — A Customizable, Language- and Front-End independent Code Analyzer", Proc. Fourteenth Int'l Conference on Software Engineering, Melbourne, Australia, May 14 – 16th, 1992.
- [9] Evans, A., Butler, K., Goos, G., Wulf, W., "DIANA Reference Manual, Revision 3", Tartan Laboratories, Inc., Pittsburgh, PA 1983.
- [10] Stroustrup, B., Internal Memorandum.
- [11] Gansner, E., North, S., Vo, K. "Dag — A Program that Draws Directed Graphs", Software – Practice and Experience, Nov. 1988, pp.1047–1062.
- [12] Merks, E. Dyck, J., and Cameron, R., "Language Design for Program Manipulation", IEEE Transactions on Software Engineering, Vol 18., No. 1, January 1992.
- [13] Teitelman, W., Masinter, L., "The Interlisp Programming Environment", Computer, 14:4 (April 1981), pp. 25-34.
- [14] Ellis, M., Stroustrup, B., The Annotated C++ Reference Manual, Addison-Wesley, 1990.
- [15] Lenkov, D., Stroustrup, B., "Run time type identification for C++", The C++ Report, March/April 1992.



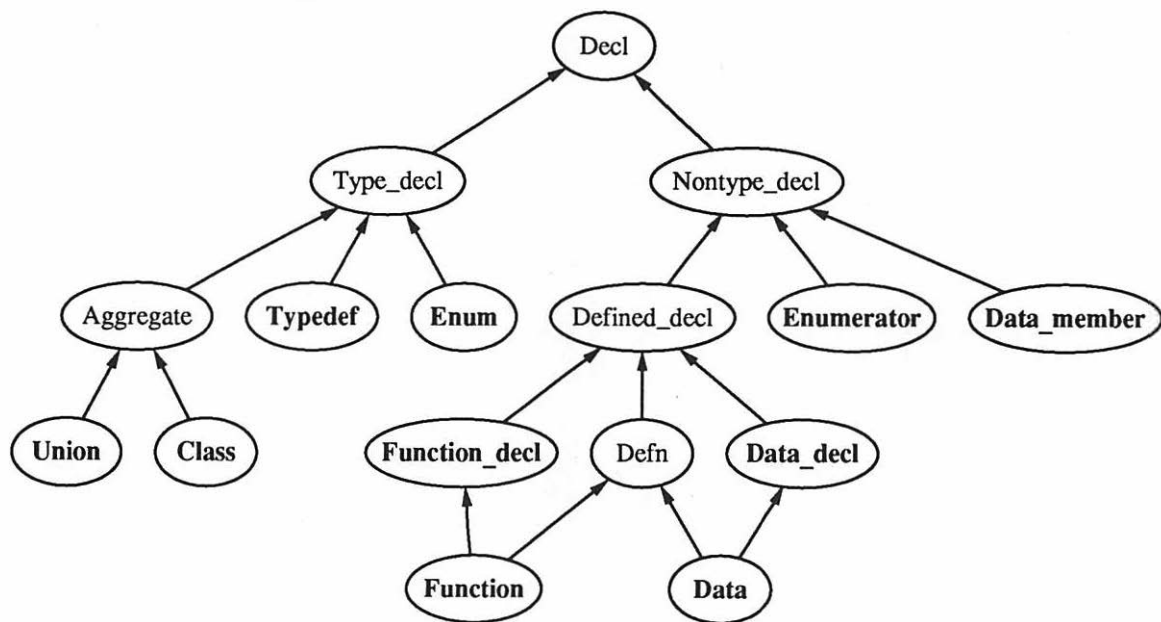


Figure 1: Declarations (inheritance hierarchy)

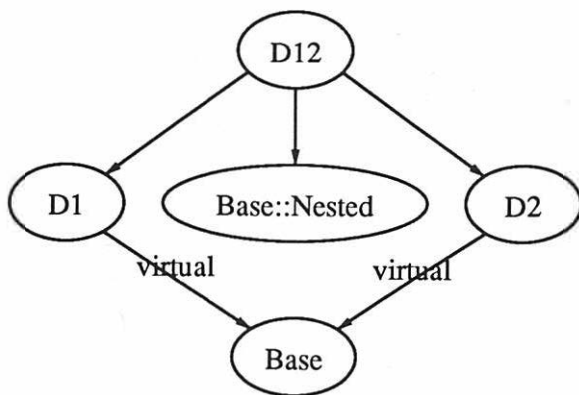


Figure 2: Dag output of the example program



# CCEL : A Metalanguage for C++

Carolyn K. Duby\*      Scott Meyers      Steven P. Reiss  
ckd@cs.brown.edu    sdm@cs.brown.edu    spr@cs.brown.edu

*Department of Computer Science  
Brown University, Box 1910  
Providence, RI 02912  
(401)863-7600*

## Abstract

C++ is an expressive language, but it does not allow software developers to say all the things about their systems that they need to be able to say. In particular, C++ offers no way to express many important constraints on a system's design, implementation, and presentation. In this paper, we describe CCEL, a metalanguage for C++ that allows software developers to express constraints on C++ designs and implementations, and we describe Clean++, a system that checks C++ code for violations of CCEL constraints. CCEL is designed for practical, real-world use, and the examples in this paper demonstrate its power and flexibility.

## 1 Introduction

C++ is an expressive language, but it does not allow software developers to say all the things about their systems that they need to be able to say. In particular, C++ offers no way to express many important constraints on a system's design, implementation, and stylistic conventions. Consider the following sample constraints, none of which can be expressed in C++:

- **Design Constraint:** *The member function  $M$  in class  $C$  must be redefined in all classes derived from  $C$ . This applies to both direct and indirect subclasses, so declaring  $M$  as a pure virtual function in  $C$  does not satisfactorily enforce the constraint.* This kind of constraint is common in general-purpose class libraries. For example, NIHCL [4] contains many such functions for the top-level Object class.
- **Implementation Constraint:** *If a class declares a pointer member, it must also declare an assignment operator and a copy constructor.* Failure to adhere to this constraint almost always leads to incorrect program behavior [12, Item 11]. A number of similar constraints was presented at last year's USENIX C++ conference [13].
- **Stylistic Constraint:** *All class names must begin with an upper case letter.* Most software development teams adopt some type of naming convention for identifiers; violations are irritating at best, confusing and misleading at worst.

---

\*Current affiliation: Cadre Technologies, Inc., 222 Richmond Street, Providence, RI 02903.

Constraints such as these exist in virtually every system implemented in C++, but different systems require very different sets of constraints. As a result, it is unreasonable to ask that C++ compilers be augmented to handle these issues. Yet the issues remain, and their importance cannot be ignored. In this paper, we describe CCEL ("Cecil") - the C++ Constraint Expression Language - a metalanguage for C++ that allows software developers to express a wide variety of constraints on C++ designs and implementations, and we describe Clean++, a system that checks C++ code for violations of CCEL constraints.

We took as our original inspiration the lint tool, which reports a number of likely error conditions in C programs. However, the errors C programmers need to detect are qualitatively different from the errors that C++ programmers need to detect. lint concentrates on type mismatches and data-flow anomalies, but type mismatches are not an issue in C++ because the language is strongly typed, and data flow analysis is unrelated to the high-level perspective encouraged by the modular constructs of C++. C++ programmers are concerned with higher-level concepts such as the structure of an inheritance hierarchy. Detection of errors in the inheritance hierarchy requires a tool that provides users with a way to check for *programmer-defined* constraints.

Other important differences between the philosophy behind lint and that behind Clean++ are those of customizability and extensibility. The set of conditions detected by lint cannot be extended by programmers, nor is there an easy way to disable the detection of classes of errors for *parts* of source files. These are significant drawbacks, and both are overcome by CCEL, as the examples in the remainder of this paper will show.

## 2 The CCEL Language

The requirements for a good constraint language are:

- The language must be powerful enough to express the constraints important to the programmer.
- The language must be intuitive and simple to learn. The look and feel must be familiar to the programmer to facilitate learning and use. Programmers need to be able to read a constraint and understand what it means in order to be able to correct a violation of a constraint, to write new constraints, and to modify existing constraints.

CCEL is based on an object-oriented model where CCEL classes represent the concepts of C++. The CCEL classes are arranged in a multiple inheritance hierarchy (See Figure 1) and have member functions defined for them (See Table 1). We determined the CCEL classes and their positions in the hierarchy by first examining in detail the concepts important to C++ programmers and the constraints they need to express. Then we classified the concepts into *CCEL classes*, such as C++ classes and member functions, and others into *properties* of CCEL classes, such as the protection level of a member function. We determined the CCEL class hierarchy by analyzing the features the CCEL classes have in common. We added abstract CCEL classes such as NamedObject to represent the common features. We chose the object-oriented model because it is familiar to users of C++,

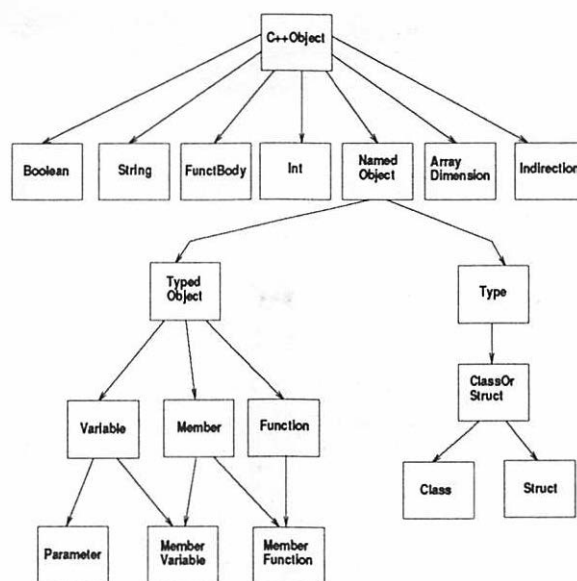


Figure 1: CCEL class Hierarchy

and because we can extend the model to add either new member functions or CCEL classes as new concepts need to be introduced.

While abstracting the concepts of C++ into CCEL classes, we often had to decide if a concept was a new CCEL class or if it could be expressed as a member function of an existing CCEL class. For example, the only difference between a class and a struct is that the default protection for a class is private, while the default protection for a struct is public. One possibility would be to put classes and structs in the same CCEL class with a boolean member function indicating whether the CCEL class is a struct. A second possibility is to put classes and structs in two different CCEL classes, with their common functionality abstracted to a base CCEL class. In general, we combined concepts into one CCEL class when the differences were trivial and the additional complexity of having a new CCEL class outweighed the increased functionality.

For example, we divided classes and structs into two CCEL classes because C++ programmers often wish to draw a distinction between them. In particular, many users believe that structs should be "just like C," while classes should be used whenever C++-specific features are employed. By separating the CCEL class concepts of classes and structs, it is straightforward to write CCEL rules that restrict the features that can be used inside structs. On the other hand, we have not yet encountered a compelling reason for differentiating between functions in general and global functions in particular (as opposed to member functions), so the current CCEL class hierarchy has no CCEL class specifically devoted to global functions. This means that there is no way to write a CCEL rule that applies only to global functions, but it would be simple enough to modify the CCEL class hierarchy if it were shown to be necessary.

CCEL constraints resemble expressions in predicate calculus, allowing the programmer to make assertions involving existentially or universally quantified CCEL variables. Clean++ reports any combination of CCEL variable values that cause the assertion to evaluate to false.

CCEL class Name	CCEL class Member Functions	CCEL class Name	CCEL class Member Functions
ArrayDimension	Int value()	MemberVariable	
Boolean	Boolean operator&&(Boolean)	NamedObject	String name()
	Boolean operator   (Boolean)	Parameter	Int position() Boolean has_default_value()
	Boolean operator!()	String	Boolean operator==(String) Boolean operator<=(String) Boolean operator>=(String) Boolean operator<(String) Boolean operator>(String) Boolean operator!=(String) Boolean matches(String)
	Boolean operator==(Boolean) Boolean operator!=(Boolean)		
C++Object	Int begin_line() Int end_line() String file()		
Class			
ClassOrStruct	Boolean is_descendant(Class)	Struct	
	Boolean is_virtual_descendant(Class)	Type	Boolean convertible_to(Type) Boolean operator==(Type) Boolean is_enum() Boolean is_union()
	Boolean is_public_descendant(Class)		
	Boolean is_friend(Class)		
FuncBody	Boolean calls(Function)		
Function	Int num_params()	TypedObject	Int num_indirections() Boolean is_pointer() Boolean is_static() Boolean is_reference() Boolean is_volatile() Boolean is_const() Boolean is_array() Boolean is_long() Boolean is_short() Boolean is_signed() Boolean is_unsigned() Type type()
	Boolean is_inline()		
	FuncBody body()		
	Boolean is_friend(Class)		
Indirection	Int level() Boolean is_const()		
Int	Boolean operator==(Int)		
	Boolean operator<=(Int)		
	Boolean operator>=(Int)		
	Boolean operator<(Int)		
	Boolean operator>(Int)		
	Boolean operator!=(Int)		
Member	Boolean is_private()	Variable	Boolean scope_is_local() Boolean scope_is_file() Boolean scope_is_class()
	Boolean is_protected()		
	Boolean is_public()		
MemberFunction	Boolean is_virtual()		
	Boolean is_pure_virtual() Boolean redefines(MemberFunction)		

Table 1: CCEL class Member Functions

Each constraint contains an assertion which must be met by some C++ source code. For example, a constraint requiring that all class names begin with a capital letter can be written in CCEL as follows:

```
// Every class name must begin with a capital letter
CapitalizeClassNames (
    Class C;      // C is a class

    Assert (C.name().matches ("^[A-Z]"));
);
```

CapitalizeClassNames is the identifier which is used to refer to the constraint. As we will see later, this identifier can be used to enable or disable the constraint. C is a CCEL variable whose domain is the set of all C++ classes in the system. The body of the CapitalizeClassNames constraint takes the form of an Assert expression, modeled loosely on the standard C assert macro facility. The assertion is that the string representing the name of the class must match the UNIX

regular expression "[A-Z]". **Class** and **Assert** are CCEL keywords; a complete list of keywords can be gleaned from the lex summary in appendix C.

As in C++, all CCEL variables must be declared before use. They are assumed to be universally quantified unless explicitly existentially quantified. Existential quantification is indicated by the use of square brackets, [...]. For example, in the constraint

```
// Every base class must have a virtual destructor.
VirtualDestInBases (
    Class B, D;

    if (D.is_descendant(B))
        Assert([MemberFunction B::f1; |
                ((f1.name() == "~{B.name()}") && (f1.is_virtual()))]);
);
```

the CCEL variables B and D are universally quantified, while the CCEL variable f1 is existentially quantified. In English, this constraint reads, "For all classes B and D, if D is a descendant of B, then it must be true that there exists a member function f1 in B such that f1's name is a tilde followed by B's name, and f1 is virtual." The legal types for CCEL variables are the CCEL classes shown in Figure 1.

CCEL variable declarations may have a condition attached to them, which is indicated by a vertical bar and a boolean expression following the variable name. For example,

```
// domain of B is all classes
Class B;

//domain of D is only classes derived from B
Class D | (D.is_descendant(B));
```

means, "for every class B and every class D such that D is a descendant of B".

By default, a constraint applies to all code in the system. This is not always desirable. For example, consider the case where a programmer has a set of naming conventions for a class library that differ from the naming conventions used for application classes. The ability to enable and disable constraint checking for named parts of the system is an important feature of CCEL. For example, if we wanted to limit the applicability of `CapitalizeClassNames` to the file "objects.C", we could declare a scope for the constraint as follows:

```
// For every class C in file "objects.C", the class name must match the
// UNIX regular expression ^[A-Z].
File "objects.C" : CapitalizeClassNames (
    Class C;

    Assert(C.name().matches("^[A-Z]"));
);
```

Sometimes it is more convenient to specify where an otherwise global constraint does *not* apply. If `CapitalizeClassNames` applies to every C++ class except `example`, we could disable `CapitalizeClassNames` for that class as follows:

```
// Do not report violations of CapitalizeClassNames in C++ class
// "lowercaseNames".
Class lowercaseNames : DontCapitalizeInLowercaseNames {
    disable CapitalizeClassNames;
};
```

Individual constraints may be grouped together into constraint classes. Suppose there are several constraints enforcing naming conventions. They could be grouped together in a constraint class called

NamingConventions as follows:

```
ConstraintClass NamingConventions {
    // For every class C, the class name must match the UNIX regular
    // expression ^[A-Z].
    CapitalizeClassNames
    (
        Class C;

        Assert (C.name().matches ("^[A-Z]"));
    );

    // For every function F, the function name must begin with
    // a lower case letter.
    SmallFunctNames
    (
        Function F;

        Assert (F.name().matches ("^[a-z]"));
    );
};
```

Notice that constraint classes are demarcated by brackets {...}, while individual constraints use parentheses (...). Constraint classes may be disabled by having a constraint such as this:

```
NamingConventionsOff (
    disable NamingConventions;
);
```

Like all CCEL constraints, this one is implicitly globally applicable. A particular constraint in a constraint class can be disabled by using the C++ scoping operator("::"):

```
SomeNamingConventionsOff (
    disable NamingConventions::CapitalizeClassNames;
);
```

If the assertion condition for a constraint is complex, the constraint designer may want to create two or more simpler constraints. The following example is a set of constraints that reports undeclared assignment operators for classes that contain a pointer member or are derived from a class containing a pointer member:



```

// If a class contains a pointer member, it must declare an assignment
// operator.
AssignmentMustBeDeclaredCond1 (
    Class      C;
    MemberVariable C::v; // v is a member variable of C

    if (v.is_pointer())
        Assert([MemberFunction C::f; |
                (f.name() == "operator=")]);
);

// If a class inherits from a class containing a pointer member, the
// derived class must declare an assignment operator.
AssignmentMustBeDeclaredCond2 (
    Class B;
    Class D | D.is_descendant(B);
    MemberVariable B::bv; // bv is a member variable of class B

    if (bv.is_pointer())
        Assert([MemberFunction D::df; |
                (df.name() == "operator=")]);
);

```

The “::” notation in the CCEL variable declaration of *v* in the first constraint indicates that *v* is a CCEL variable whose domain is the member variables of class *C*. To be a member of a class means that a member variable or member function is declared in this class, i.e. is not inherited. The `redefines` member function of the Class CCEL class (see Figure 1) can be used to find out if a class defines a function with the same parameters and name as the given member function, i.e. if C++ would view it as a virtual redefinition. In the first constraint, the existentially quantified CCEL variable *f* is used to determine if an assignment operator is defined for classes containing a pointer member variable. In the second constraint, the existentially quantified variable *df* is used to check if the descendants of a base class containing a pointer member define an assignment operator.

The Member, MemberVariable, and Parameter CCEL classes have special conditions attached to them similar to the MemberFunction object. A :: in a MemberVariable or Member variable declaration separates the class name from the member variable name. Examples:

```

Class C;           // C is a class
Member C::M1;      // M1 is a member of class C
Function      F;   // F is a function
Parameter     F(P); // P is a parameter of F
Variable      V;   // V is a variable with local, file, or
                  // class scope

```

Any combination of the C++ relational and logical operators can be used inside Assert clauses. As with C++, only types for which comparison has been defined can be used in comparison operations. The types of the items being compared must be the same, and the arguments to the boolean predicates must be boolean expressions.

### 3 Error Messages

Since constraints are user-defined and not hardcoded, the CCEL evaluator cannot form a meaningful error message describing what the error condition is. The best the evaluator can do is print the constraint violated and the current values of the CCEL variables that violated the assertion. Therefore, we decided to allow the user to optionally associate a message to be reported with every constraint. The programmer-defined error messages enables the user to word the error message in familiar terms and also allows users who do not know CCEL to use Clean++ to find program errors. If CCEL did not allow users to define their own error messages, the only way for a user to determine the error condition would be to interpret the assertion and understand what needs to be changed to correct the violation. This would force every user of Clean++ to be proficient in CCEL.

An error message is an interpreted string which may contain references to bound CCEL variables and to the builtin variables `ConstraintId`, `ConstraintFile`, and `ConstraintLine` which print the unique identifier, file, and line that the constraint violated is defined at. Variables are denoted by { ... }. For instance, when using NIHCL[4], a programmer would want require that every class derived from "Object" declare the `isA` function with the constraint:

```
// The member function Object::isA must be redefined in all
// subclasses of class Object
RedefineisA(
    Class B | (B.name() == "Object");
    Class D | (D.is_descendant(B));
    MemberFunction B::f1 | (f1.name() == "isA");

    Assert({MemberFunction D::f2; | f2.redefines(f1)});
) "{D.file()}{line {D.begin_line()}} constraint {ConstraintId} in
\"{ConstraintFile}\" : \n
Class {D.name()} does not define function isA.";
```

If class `MySubclass` does not declare the `isA` function, the violation message reported is:

```
mysubclass.h(Line 5) constraint RedefineisA in "constraints.ccel" :
    Class MySubclass does not define function isA.
```

If the programmer does not define an error message for the constraint, a standard error message showing the value of the CCEL variables that caused the assertion to be violated and the identifier, line number and file of the constraint violated is reported. The default error message for the previous example is :

```
constraint RedefineisA in "constraints.ccel" (line 36):
    Class B = Object on line 20 of file "Object.h"
    Class D = MySubclass on line 5 of file "MySubclass.h"
    MemberFunction f1 = isA on line 23 of file "Object.h"
```

### 4 Prototype Architecture

There are two possible approaches to implementing Clean++. The first approach takes the constraints and generates a custom program that reads the C++ source and checks for violations of the

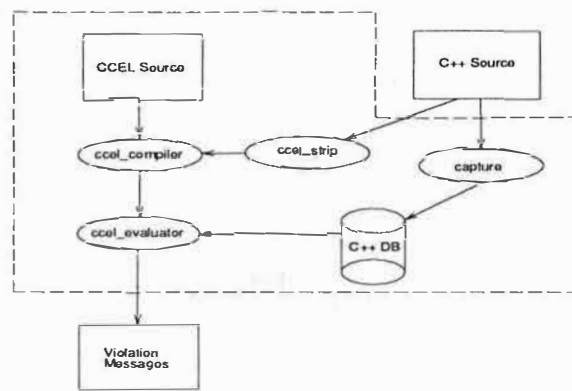


Figure 2: Clean++ Architecture

constraints. The second approach extracts data from the C++ program and stores it in a database. The constraints are then converted into queries over the database. If a constraint is violated, its corresponding query will have a non-null result containing information about the violations.

The first approach requires generation of a custom constraint-checking program for each set of constraints, but for the second approach, a single constraint-interpreter suffices. As a result, we chose the second approach, because only one program is needed to apply multiple sets of constraints to the same system.

Figure 2 shows the components of Clean++. Clean++ constraints can be specified in one or more files or within a C++ program by embedding them in C++ comments. (This is similar to the way lint error messages can be controlled from within C source.) All features of CCEL can be used inside the C++ source, but we expect programmers will use it primarily to specify constraints specific to a class or file within the C++ source, i.e. to associate a constraint with the C++ source to which it applies. By associating constraints with C++ source, developers who want to derive a new class from a base class or create an instance of a class can see any constraints that apply to a class. For example, a constraint stating that all subclasses of a class must redefine a particular member function would be best put in the C++ source file for the class so that programmers know that they will need to define that member function. A more generic constraint, such as every class name must begin with an upper case letter, might go in a file containing style constraints.

The Clean++ architecture is made up of the following independent components: `ccel.strip`, `ccel.compiler`, `ccel.evaluator`, and `capture`. The `ccel.strip` program is a simple preprocessor that searches the C++ source files and extracts any constraints embedded in comments. The constraints from the C++ source and any constraint files are passed to `ccel.compiler`, which uses the C preprocessor to process any macros or file inclusions. `ccel.compiler` then translates the constraints into queries over the C++ database. The `ccel.evaluator` program executes the queries and outputs any violations of the constraints.

The `capture` program parses the C++ files and places their semantic structure in the C++ repository. There are several existing systems that can capture the structure and semantics of C++ programs. Such systems include REPRISE[14], CIA++[5], and XREFDB[7]. Of these, REPRISE seems most suited for use with CCEL, and our prototype implementation uses REPRISE for the

database portion of Clean++.

For comparison purposes, the dotted lines in Figure 2 enclose the functionality of the lint tool for C programs. As is clear from the diagram, Clean++ gives the programmer more control by allowing the programmer to access the inner architecture of the checker and customize which constraints are checked. In particular, CCEL users can modify the CCEL source (i.e., the set of constraints to be enforced), while users of lint are unable to modify the conditions detected by lint.

## 5 Related Work

Support for formal design constraints in the form of assertions or annotations was designed into Eiffel [10], has been grafted onto Ada in the language Anna [9], and has been proposed for C++ in the form of A++ [2, 1]. This work, however, has grown out of the theory of abstract data types [8], and has tended to limit itself to formally specifying the semantics of individual functions and/or collections of functions (e.g., how the member functions within a class relate to one another). CCEL has a different focus. It has little concern for the semantics of functions;<sup>1</sup> however, it allows programmers to express constraints involving virtually any kind of declaration. As such, it is able to constrain relationships between classes, which Eiffel, A++, and Anna are unable to do. CCEL can also express constraints on the concrete syntax of C++ source code (e.g., CCEL class-specific naming conventions); this is also outside the purview of semantics-based constraint systems.

GENOA [3] is a language-independent application generator that can be used to generate a wide variety of code analysis tools. GENOA specifications consist of actions to be performed at nodes of an attributed parse tree. Unlike CCEL, which is specifically designed for C++ *programmers*, GENOA is designed for *compiler writers*. GENOA users must know the structure of the programming language parse tree, because applications based on GENOA are actually custom-designed traversals of this tree. CCEL hides this kind of grammatical detail, and is hence much easier to learn and use. The downside, of course, is that CCEL cannot be as expressive as applications taking full advantage of the generality of the GENOA approach.

## 6 Status

To date, our work on Clean++ has focused on developing a workable architecture for the system and on the design and implementation of CCEL. We have implemented a parser for CCEL and have completed the bulk of semantic analysis. Currently, we are concentrating on the implementation of a constraint evaluator for the internal representation of CCEL constraints; we are using REPRISE as the source of information about C++ source code. We expect to have a fully functional prototype for Clean++ before this research is presented at the USENIX C++ conference.

---

<sup>1</sup>In fact, the current version of CCEL cannot express anything at all about function *definitions*. However, enhancing it so that it can is the next logical extension to the language.

## References

- [1] Marshall P. Cline and Doug Lea. The Behavior of C++ Classes. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 81–91, September 1990.
- [2] Marshall P. Cline and Doug Lea. Using Annotated C++ . In *Proceedings of C++ at Work - '90*, pages 65–71, September 1990.
- [3] Premkumar T. Devanbu. GENOA – a customizable, language- and front-end independent code analyzer. In *Proceedings of the International Conference on Software Engineering*, May 1992.
- [4] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.
- [5] Judith E. Grass and Yih-Farn Chen. The C++ Information Abstractor. In *USENIX C++ Conference Proceedings*, pages 265–277, 1990.
- [6] Moises Lejter, Scott Meyers, and Steven P. Reiss. Adding Semantic Information To C++ Development Environments. In *Proceedings of C++ at Work-'90*, pages 103–108, September 1990.
- [7] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for Maintaining Object-Oriented Programs. In *Proceedings of the Conference on Software Maintenance*, October 1991. This paper is largely drawn from two other papers [11, 6].
- [8] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.
- [9] D. Luckham, F. von Henke, B. Krieg-Bruckner, and O. Owe. *Anna, A Language for Annotating Ada Programs: Reference Manual*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.
- [10] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science. Prentice Hall, 1988.
- [11] Scott Meyers. Working with Object-Oriented Programs: The View from the Trenches is Not Always Pretty. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 51–65, September 1990.
- [12] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1992.
- [13] Scott Meyers and Moises Lejter. Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++. In *USENIX C++ Conference Proceedings*, pages 29–40, April 1991.

- [14] David S. Rosenblum and Alexander L. Wolf. Representing Semantically Analyzed C++ Code with Reprise. In *USENIX C++ Conference Proceedings*, pages 119 – 134, April 1991.

## A Additional Examples

The constraints that follow supplement the examples given in the body of the extended abstract. They serve to help demonstrate not only the expressiveness of the CCEL language itself, but also the kinds of constraints that C++ programmers might well want to enforce.<sup>2</sup>

```
// Subclasses must not redefine an inherited non-virtual member
// function.
NoNonVirtualRedefines (
    Class B, D;
    MemberFunction B::mb;
    MemberFunction D::md;

    if (D.is_descendant(B) && md.redefines(mb))
        Assert(mb.is_virtual());
) "Class {B.name()} redefines inherited non-virtual member
function {mb.name()}.";

// The return type of the assignment operator must be a reference to
// the class
ReturnTypeOfAssignmentOp (
    Class C1;
    MemberFunction C1::m1;

    if (m1.name() == "operator=")
        Assert(((m1.is_reference()) && (m1.type() == C1)));
) "The assignment operator for class {C1.name()} does not return a
reference to class {C1.name()}.";

// If a class contains a pointer member, the copy constructor must
// be defined.
CopyConstructorDefined (
    Class C;
    MemberVariable C::v1;

    if (v1.is_pointer())
        Assert(
            [ MemberFunction C::f1;
              Parameter    f1(p1); |
              ((f1.name() == C.name()) && (f1.num_params() == 1) &&
               (p1.type() == C) && (p1.is_reference()))]);
) "A copy constructor should be defined for class {C.name()} because it
contains the pointer member {v1.name()}";
```

---

<sup>2</sup>Although the grammar does not allow newlines in error message text, the error messages were separated for presentation purposes.

```

// Members should be declared in the order public, protected, private
MemberDeclOrdering (
Class C;

Assert(!([Member C::pub_mem | (pub_mem.is_public());
        Member C::non_pub_mem | (!non_pub_mem.is_public()); |
        (pub_mem.begin_line() > non_pub_mem.begin_line())]) ||

        [Member C::prot_mem | (prot_mem.is_protected());
        Member C::priv_mem | (priv_mem.is_private()); |
        (prot_mem.begin_line() > priv_mem.begin_line())]) ||

        [Member C::priv_mem | (priv_mem.is_private());
        Member C::non_priv_mem | (!non_priv_mem.is_private()); |
        (non_priv_mem.begin_line() > priv_mem.begin_line())]);

) "Class {C.name()} has members that are not declared in the order
public, protected, private.";

// Derived classes should not redefine an inherited default parameter
// of a virtual function.
NoRedefineOfDefaults (
Class B, D;
MemberFunction D::df, B::bf;
Parameter df(p1), bf(p2);

if ((D.is_descendant(B)) && (df.is_virtual()) &&
    (df.redefines(bf)))
    Assert(((p1.position() == p2.position()) &&
            (p1.has_default_value() == p2.has_default_value())) ||
            (p2.position() != p1.position()));
) "Member function {D.name()}::{df.name()} redefines member function
{B.name()}::{bf.name()} but parameters {p1.name()} of function
{df.name()} and {p2.name()} of function {bf.name()} have different
defaults.";

// Multiple inheritance hierarchies should not be diamond-shaped.
HierarchyStructure (
Class A, B1, B2, C;

Assert(!((B1.is_descendant(A)) && (B2.is_descendant(A)) &&
        (C.is_descendant(B1)) &&
        (C.is_descendant(B2)) && (B1.name() != B2.name())));
) "Class {A.name()} is an ancestor of {B1.name()} and {B2.name()} which
are ancestors of {C.name()}." ;

```

## B CCEL Grammar

What follows is the YACC grammar for the CCEL prototype; semantic actions have been excluded.

```
%union {
    char      *cval;
    int       ival;
}

%token <cval> STRING IDENT CONSTRAINT_CLASS_KEY CLASS_KEY FILE_KEY
%token <cval> C_PLUS_PLUS_OBJECT_KEY FUNCTDEF_KEY TYPEDOBJECT_KEY
%token <cval> PARAMETER_KEY MEMBERVARIABLE_KEY MEMBERFUNCTION_KEY
%token <cval> ENABLE_KEY DISABLE_KEY IF_KEY MEMBER_KEY VARIABLE_KEY
%token <cval> FUNCTION_KEY ARRAY_DIM_KEY NAMEDOBJECT_KEY
%token <cval> CLASSORSTRUCT_KEY STRUCT_KEY TYPE_KEY ASSERT_KEY
%token <cval> INTEGER INDIRECTION_KEY
%type <cval> var_name
%%

constraint_file : constraint_file constraint_group |
                constraint_group ;

constraint_group : constraint_class |
                 constraint_list ;

constraint_class : CONSTRAINT_CLASS_KEY constraint_class_ident '{'
                 constraint_list '}' ';' ;

constraint_class_ident : unique_id ;

constraint_list : constraint_list constraint |
                constraint ;

unique_id : IDENT ;

constraint : opt_constraint_scope constraint_ident '('
            constraint_body ';' ')'
            opt_message ';' ;

constraint_body : variable_decls constraint_condition |
                select_constraint ;

constraint_ident : unique_id ;

opt_constraint_scope : scope_obj obj_name ':' |
                    ;

scope_obj : CLASS_KEY | FILE_KEY | FUNCTION_KEY | VARIABLE_KEY ;

obj_name : filename |
          IDENT ;
```



```

filename : STRING ;

opt_message : STRING |
;

variable_decls : variable_decls variable_decl |
;

variable_decl : type var_name_list optional_cond ';'
;

type : C_PLUS_PLUS_OBJECT_KEY |
      FUNCTDEF_KEY |
      TYPEDOBJECT_KEY |
      TYPE_KEY |
      VARIABLE_KEY |
      MEMBER_KEY |
      FUNCTION_KEY |
      CLASS_KEY |
      PARAMETER_KEY |
      MEMBERVARIABLE_KEY |
      MEMBERFUNCTION_KEY |
      ARRAY_DIM_KEY |
      NAMEDOBJECT_KEY |
      INDIRECTION_KEY |
      CLASSORSTRUCT_KEY |
      STRUCT_KEY;

var_name_list : var_name_list ',' var_name |
               var_name ;

var_name : IDENT ':' IDENT |
          IDENT '(' IDENT ')' |
          IDENT '[' IDENT ']' |
          IDENT;

optional_cond : '|' expression |
;

constraint_condition :
    IF_KEY '(' expression ')' ASSERT_KEY '(' expression ')' |
    ASSERT_KEY '(' expression ')'
;

param_list : expression_list |
;

expression_list : expression_list ',' expression |
                expression;

expression : simple_expression |
            simple_expression '=' simple_expression |
            simple_expression '!' '=' simple_expression |
            simple_expression '<' simple_expression |

```

```

        simple_expression '>'      simple_expression |
        simple_expression '<' '=' simple_expression |
        simple_expression '>' '=' simple_expression ;

simple_expression :
    term |
    simple_expression '|' '|' term;

term : factor |
    term '&' '&' factor;

factor : STRING |
    INTEGER |
    IDENT function_list |
    '!' factor |
    '(' expression ')' |
    '[' variable_decls '|' expression ']' ;

function_list : function_list '.' IDENT '(' param_list ')' |
    ;

select_constraint : on_or_off constraint_selector ;

on_or_off : ENABLE_KEY |
    DISABLE_KEY ;

constraint_selector : selected_constraint |
    constraint_class_ident ;

selected_constraint : constraint_class_ident ':' ':' constraint_ident;

```

## C CCEL Tokens

What follows is the LEX source for the CCEL prototype.

```

ws      [ \n\t]
letter  [A-Za-z]
digit   [0-9]
integer {digit}+
punct   [_]
%%

{ws}                {skip_whitespace();}
{integer}            {sscanf(yytext, "%d", yylval.ival);
                     return(INTEGER);}

"//"                {skip_comment();}
"ArrayDim"           {return(ARRAY_DIM_KEY);}
"ConstraintClass"     {return(CONSTRAINT_CLASS_KEY);}
"Class"              {return(CLASS_KEY);}
"File"               {return(FILE_KEY);}
"Function"           {return(FUNCTION_KEY);}

```

```

"Variable"                { return(VARIABLE_KEY); }
"C++Object"               { return(C_PLUS_PLUS_OBJECT_KEY); }
"FuncBody"                { return(FUNCTBODY_KEY); }
"TypedObject"             { return(TYPEDOBJECT_KEY); }
"Type"                    { return(TYPE_KEY); }
"Member"                  { return(MEMBER_KEY); }
"Parameter"               { return(PARAMETER_KEY); }
"MemberVariable"          { return(MEMBERVARIABLE_KEY); }
"MemberFunction"          { return(MEMBERFUNCTION_KEY); }
"NamedObject"             { return(NAMEDOBJECT_KEY); }
"Indirection"             { return(INDIRECTION_KEY); }
"ClassOrStruct"           { return(CLASSORSTRUCT_KEY); }
"Struct"                  { return(STRUCT_KEY); }
"Assert"                  { return(ASSERT_KEY); }
"if"                      { return(IF_KEY); }
"enable"                  { return(ENABLE_KEY); }
"disable"                 { return(DISABLE_KEY); }
{letter}({letter}|{digit}|{punct})*
                           {yylval.cval =
                             new char[strlen(yytext)+1];
                             strcpy(yylval.cval, yytext);
                             return(IDENT); }
\".*\"                    {yylval.cval =
                           new char[strlen(yytext)+1];
                           strcpy(yylval.cval, yytext);
                           return(STRING); }
                           {return(yytext[0]); }

```



# Space-efficient trees in C++

Andrew Koenig

*AT&T Bell Laboratories*

*600 Mountain Avenue; Murray Hill NJ 07974; ark@europa.att.com*

## Context

Programmers often find it useful to store data in tree form. Sometimes they use trees to make searching faster (such as in the many kinds of search trees); other times they use the structure of the tree to model the structure of their data (such as parse trees in a compiler).

When programs of the latter kind have a lot of data to store, space efficiency may become more important than time efficiency. If there is not enough memory available to store a very large tree, it doesn't matter that it can be traversed quickly because that speed will be lost to paging activity anyway.

Consider a programming environment intended for interactive development of potentially very large systems. How might such an environment store the programs under development? In a sense, it is most direct simply to store source text. However, source code does not directly express anything about the actual structure of the program; substantial analysis is often necessary to determine that structure.

For example, it is easy to imagine a user of such an environment making some query that requires generating the complete call graph of a system. The only way to obtain that call graph is to analyze the entire system, which for large systems can be very time consuming.

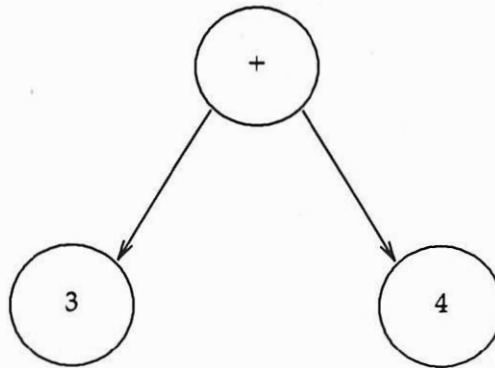
The Grail environment kernel for C++, presently under development at AT&T Bell Laboratories, starts with the assumption that it is worthwhile to keep around the results of that analysis in place of conventional source text. That means that programs are stored in a way that reflects their underlying structure. Not surprisingly, that representation is largely comprised of trees, for the same reason that the syntax of programming languages is often described by context-free grammars.

Of course, trees are useful for things other than programming environments. For example, most modern operating systems implement tree-structured file systems. It is also not unusual to impose tree structures on documents to make them easier to understand.

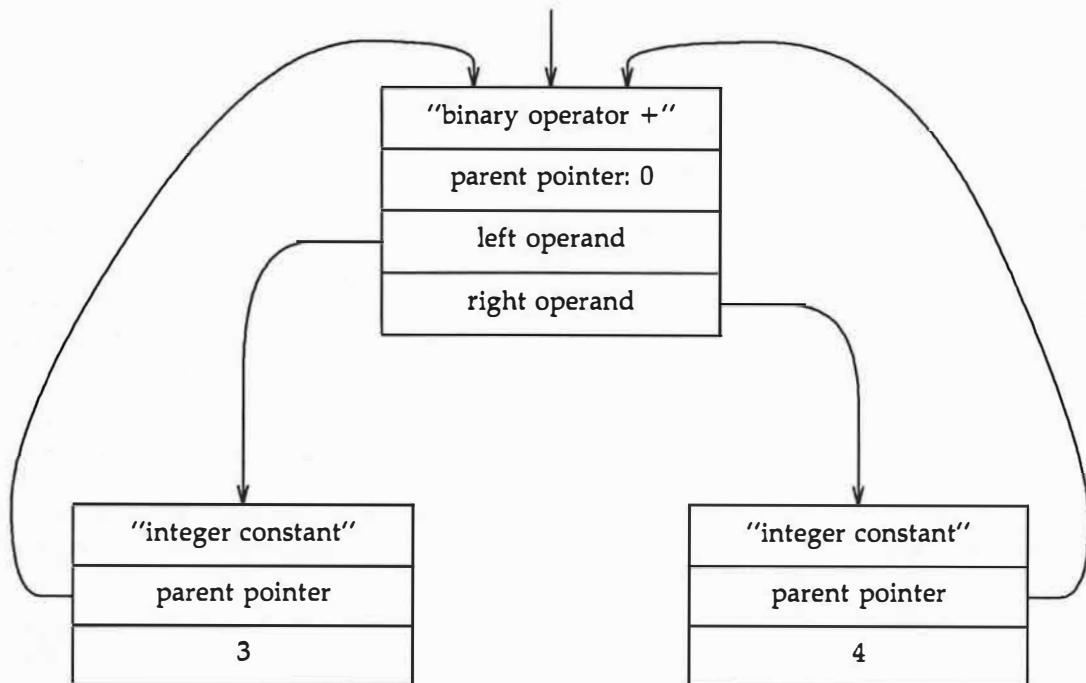
## The problem

When drawing a tree, one usually includes with each node only pointers to its children and auxiliary information, often called a *label*. Conceptually, each node in a tree contains only a label and pointers to its children. In a language like C, the usual way to implement such a tree is to store an additional piece of information, called a *type code*, in each node. Loosely speaking, there is one type code for each kind of node; inspecting the type code makes it possible to know the type (in the C sense) of the information, if any, stored in the node. The type code also determines how many children the node has and where in the node to find the pointers to its children. Additionally, it can make tree traversals much easier if every node contains a pointer to its parent (To see this, imagine the job of a programming environment that has somehow found the piece of code that its user is looking for and now wants to display the context that surrounds it).

For example, the following simple tree might conceptually represent the expression  $3+4$ :



but the typical implementation of that tree would look like this:



Each node begins with a type code followed by a parent pointer. The root of the tree has a zero parent pointer; of course this would change if the tree were part of some larger tree.

Let's pause at this point and see if we can estimate how much memory this tree representation will use. We will make the realistic assumption that our machine allocates memory a word at a time, where a word is the size of an `int` or a pointer (four bytes on most machines large enough to support the kind of programming environment we're discussing). This assumption is not valid for all machines, but it is a good basis for an estimate. Assume also that there is no additional overhead for the memory allocator itself: it should be possible to design a special-purpose allocator that uses the type of a node to figure out how big it is when it comes time to free it.

The root of our tree must store the type code, the pointer to its parent, and pointers to its two children. Thus the `+` node must contain at least four words.

The nodes for `3` and `4` must similarly use three words each: one for the type code, one for the parent pointer, and one for the value of the constant itself.

Our naive representation of 3+4 therefore consumes ten words, of which five are pointers. In other words, in this example half of our memory is devoted to pointers.

## Strategy

Each node contains two parts: pointers and other information. The pointers correspond to the structure of the intuitive tree; the other information corresponds to the node labels. Because the two parts have such different purposes, we are likely to need different strategies to reduce the memory requirements of each part.

If the parts were grossly different in size, we would be able to concentrate on the bigger one. Unfortunately, our examples show that the two parts of our nodes are about the same size, which suggests that we need a way to shrink both of them.

With the benefit of hindsight, we will first show how to reduce the data requirements of each node by exploiting the fact that many nodes will have the same type and value as other nodes. Once that is done, we will be able to exploit the regularity of the resulting structure to reduce the storage required for the tree data structure itself.

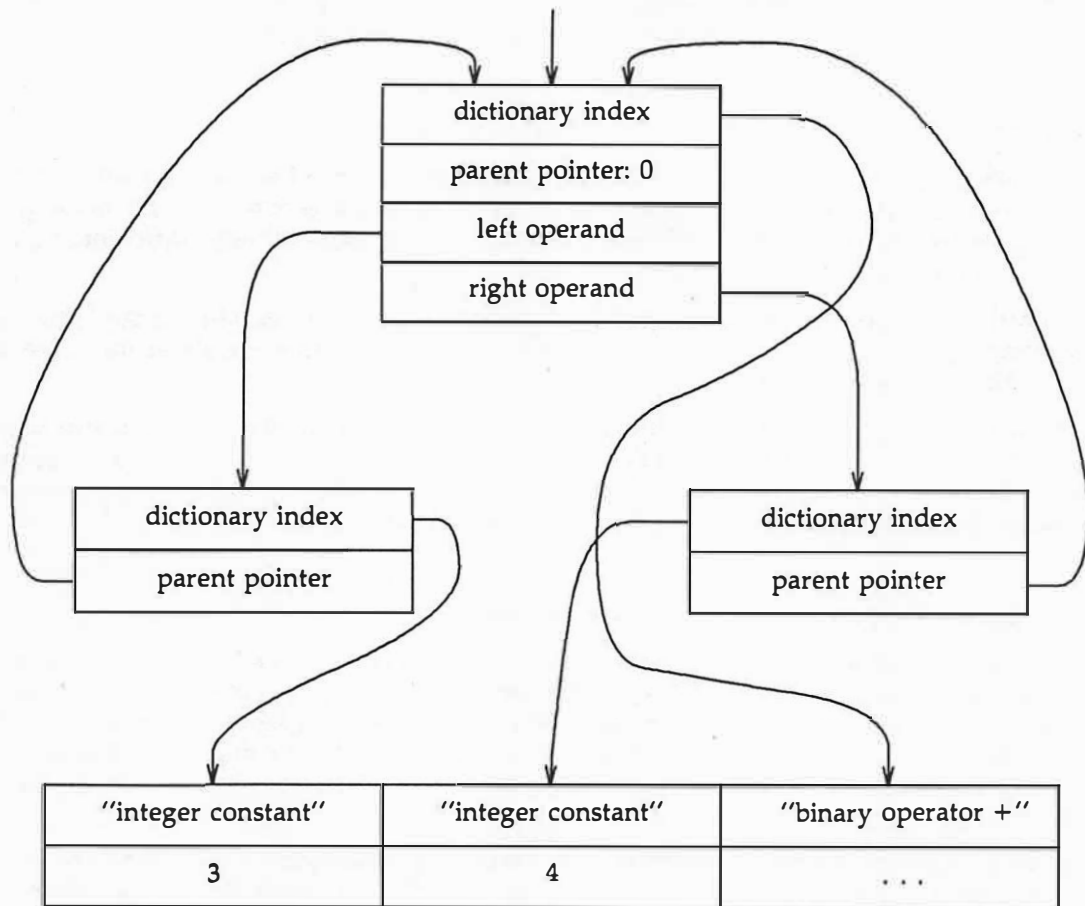
## A dictionary removes some of the overhead

Most programs mention identifiers several times if they mention them at all. For example, a program that declares a variable will probably use it as well, a program that uses an operator in one place is likely to use it again elsewhere, and so on. We may therefore be able to remove the memory required for the data part of each node by moving the entire data part into a separate dictionary in which each unique entity (identifier, operator, etc.) appears only once.

In practice, a programming environment may need to maintain such a dictionary anyway. If we have already paid for space in the dictionary, using that space for two purposes costs nothing extra. Even if we haven't, using a dictionary means that the space occupied by the value is amortized over all the times that value appears.

Once we have such a dictionary, we can store the node types in the dictionary along with their values. For the nodes that don't have values, we can create dummy values in the dictionary whose only purpose is to hold the type. We need only one such value per type, so the overhead for doing that is fixed and small. This makes it unnecessary to store types in tree nodes at all; all we need are dictionary indices.

Using this technique makes our tree look like this:



Dictionary

If we assume we're going to need a dictionary anyway, this technique reduces the size of the nodes in our example: the root still requires four words but each leaf now requires only two. Thus the total size of our tree has shrunk from ten words to eight. However, five of those words are still pointers, so the fraction of space occupied by pointers has gone from half to five-eighths. Is there any way to reduce the space needed by those pointers? Wait and see.

### The dictionary has made the nodes more regular

By using a dictionary, we have made our nodes fairly uniform in structure. The only ways in which individual nodes differ from each other is

- Nodes may have different numbers of children.
- Each node has a type code and possibly a value, both of which are stored in the dictionary. The type determines the number of children.
- Even if two nodes refer to the same place in the dictionary, they are still different nodes. That is, if our 3+4 expression were 3+3, it would be possible to determine that the two leaves are different nodes even though each one contains 3.

In other words, by putting the values and type codes into the dictionary, we can ignore the whole issue of node types! From now on, all we need to know about a node is

- How many children it has, which we will call its *valence*;



- Its dictionary offset, which we will call its *value*; and
- Its identity.

## What things must we do quickly?

In software design it is classic to trade time for space. We give up the ability to do some operations quickly—and sometimes to do them at all—in exchange for a more compact representation. Which operations are important to do quickly in a programming environment? Perhaps more importantly, which ones are not?

One fundamental decision in Grail is that it is not important to be able to modify the representation of a program quickly. In fact, it is not clear that we wish to allow destructive modification at all! This is true for several reasons:

- Not every change is what its author intended. A very easy way to allow programmers to undo changes they've made is simply to keep the old version around and let them revert to it. This can consume a lot of space if "the old version" is a complete copy of the source text, but it may be quite sensible if we need to replicate only the subtree that has been changed.
- Keeping previous versions of programs around makes version control much easier.
- Even if we do decide to discard old versions of things, programs are read much more often than they are changed. Thus it seems to make sense to make storage more efficient at the cost of making it harder to change things.

It is clearly important to have efficient operations that somehow visit every node of a tree. Something like this is necessary to allow searching a program for references to a particular variable, for example, and probably for input-output as well.

It must also be efficient to do the things for which one would normally use the pointers in a tree directly: move from a node to one of its children or its parent. However, it may be acceptable for those operations to be somewhat slower if traversals are fast.

## Introducing the Shrub class

What do we know about the problem so far? We want to be able to represent trees in a way that makes them fast to traverse and search and reasonably fast to navigate. We don't care about changing a tree after it has been built. Moreover, we have already seen that the information in each node is limited to valence, value, and identity. How might we use this information to reduce overhead?

If we do not care about changing a tree once built, that implies that each tree goes through two distinct phases during its existence. First it will be built, during which time we do not care about being able to traverse it. When we're done building it, we will say so, after which we can traverse it but we must not change it further.

This observation suggests something about a natural interface to a class for compact storage trees. We can therefore begin by writing a C++ class that incorporates just this suggestion. Because the purpose of this exercise is compact storage of trees, we will call our class a *Shrub*.

We assume, then, that we will say something like this:

```

Shrub<T> s;

// build up the Shrub, then ...

s.done();

// Now s is assumed to be fully built;
// we can't change it any more.

```

Several decisions are implicit even in this tiny example. We have taken advantage of one of our earlier constraints, namely that all the nodes contain values of the same type. That allows us to define `Shrub<T>` as a template and talk about techniques for dealing with shrubs without reference to the particular type being stored. We will think of `T` as being the type of a dictionary entry, but we will not really make any assumptions about `T` at all; that's up to whoever uses this class.

How should one go about building a shrub? A straightforward way is simply to present the nodes in some canonical order. Suppose, for example, that we present the nodes in preorder. Because the node's value determines its valence, that presentation is actually sufficient to determine the structure of the tree, just as parentheses are unnecessary to determine the structure of an expression represented in Polish notation.

However, it turns out that in the context of Grail, it is sometimes inconvenient to determine the value of the root of a subtree until after its leaves have been determined. We can solve that problem by having a separate member function to mark the *beginning* of each subtree and then presenting the actual node values in *postorder*.

In other words, building a non-trivial subtree involves first calling a particular member function, next doing whatever it takes to build the body of the subtree, and finally supplying the value for the root of that subtree. Building a leaf, of course, requires merely supplying an appropriate value for that leaf.

Thus, to build a shrub representing `3+4`, we first say that we are about to build a subtree, then supply the value for `3`, then the value for `4`, and finally the value for `+`. The code to do that might look like this:

```

Shrub<T> s;

s.begin_subshrub();
s.addnode(three);
s.addnode(four);
s.addnode(plus);
s.done();

```

Here, of course, `three`, `four`, and `plus` are the dictionary indices for `3`, `4`, and `+`.

The call to `begin_subshrub` indicates that we are about to build an interior node, the value of which will be supplied later. The next two calls to `addnode` supply the values of the two leaves. Finally, we supply the root of the subshrub and say we're finished.

The `Shrub` class "knows" that `three` and `four` represent leaves and `plus` does not, but only if we tell it. To be able to use `Shrub<T>`, we must define a function called `valence` that takes a value of type `T` and returns an integer representing the number of children of a node of that value. Thus we must define `valence` so that `valence(three)` and `valence(four)` are both `0` and `valence(plus)` is `2`.<sup>1</sup>

## Climbing around in trees

It is all very well to be able to construct a `Shrub` with given contents, but that is useless unless we have some way to get information out again. Moreover, that way must be independent of the representation actually used for the data; we cannot simply hand our users pointers to the nodes because that would reveal too much about the implementation. The usual C++ solution to this is to define a class to encapsulate the important parts of the idea we want to use. What ideas about pointers are important?

A pointer enables us to mark a particular node of a tree. We can use pointers to tell whether two nodes are actually the same: pointer identity is equivalent to node identity. We can also fetch the node associated with a pointer and use the pointer to find other nodes related to the one pointed to.

Since a pointer marks a spot in a tree, we'll call our pointer-like class a `Tag`. A `Tag` can be thought of as being *attached* to a particular node of a particular `Shrub`. `Tags` should support other pointer operations: it should be possible to copy them, pass them to functions, return them as results, compare them, and so on. In addition, we would like to use `Tags` for navigation: once we have a `Tag` on a particular node, we should be able to move that `Tag` to a related node, such as a child, sibling, or parent.

How might these operations work? If `s` is a `Shrub<T>`, as before, then the simplest way of creating a `Tag<T>` associated with it is to say

```
Tag<T> t(s);
```

That presumably says that the `Tag` should be attached to some node of the `Shrub`. Every `Shrub` has a root, so we can conveniently attach the `Tag` there.

How do we get at the element marked by `t`? If a `Tag` acts like a pointer, then `*t` might appear to do the trick. However, that would require each `Tag` to know the identity of its associated `Shrub`. Since large numbers of `Tags` may well be stored in large quantities, it may well make more sense to treat them like subscripts and require the user to supply the identity of the `Shrub` as well,<sup>2</sup> so `s[t]` is the node in the `Shrub` called `s` that is marked by the `Tag` called `t`.

We can use a similar approach to define other `Tag` operations:

<code>s.move_down(t,n)</code>	Move <code>t</code> , which is attached to a node in <code>s</code> , to the <code>n</code> th child of that node, if that child exists.
<code>s.move_up(t)</code>	Move <code>t</code> to its parent, unless <code>t</code> is already at the root.
<code>s.advance(t)</code>	Move <code>t</code> to the next node in the preorder traversal of <code>s</code> , unless <code>t</code> is already at the last node.

In principle it should be easy to implement a wide variety of `Tag` operations to allow people to wander around in `Shrubs` as they wish.

- 
1. The actual implementation allows each `Shrub` object to have its own dictionary. That means that a given value may have a different valence in different `Shrubs`. To allow that, `valence` actually takes two arguments: a `T` and a `Shrub*`. Later versions have changed that to a `Shrub&`.
  2. Why require the user to supply the identity of the `Shrub` instead of implementing a `Tag` as a pointer and extracting the `Shrub` identity from it? The main reason is that locating a `Shrub` from the address of one of its elements may not be possible quickly and we need that information often, particularly for use as an argument to the user's `valence` function.

## Theory and practice

It is time to think about getting rid of the remaining overhead in the nodes, namely the pointers that represent the tree structure. It's hard to imagine reducing the overhead any more than eliminating it entirely, so that is a logical place to start.

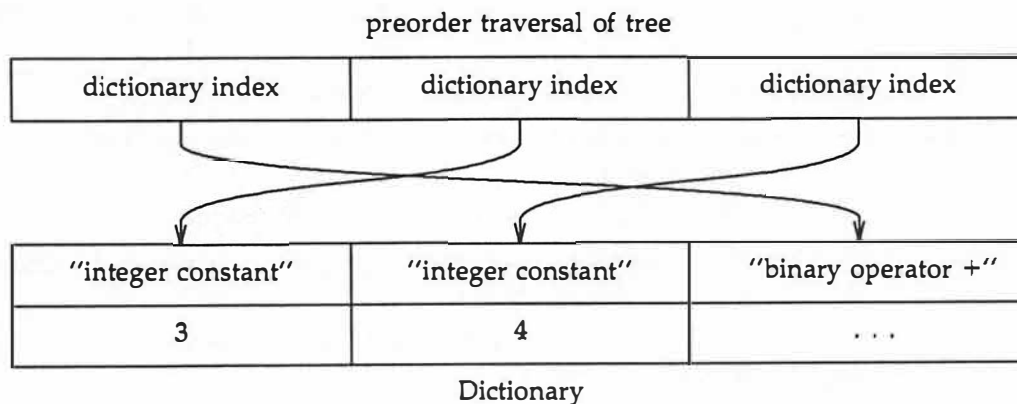
Suppose we could eliminate all the pointers from a tree. All that would be left would be the node values themselves. Fortunately, we still have a little auxiliary information available: if the nodes are stored in some sequence (as, for instance, elements of an array inevitably are), perhaps that sequence alone is all the information we need.

For example, if we traverse the tree in preorder and store the sequence of nodes that results from the the traversal, that sequence is enough to reconstruct the original tree (References: Knuth, Read). One might imagine an implementation of the Shrub class that lets its users compress and expand the representation so that the tree is stored in compact form when it is not in use.

The trouble with that is that we still need enough memory to store the reconstituted tree, which has all the overhead of the original tree. If the tree is large, or if many trees are potentially in use at once, this raises the question of when to discard or recompress the reconstituted trees. A more interesting question is therefore whether it is possible to use the tree without reconstituting it. In other words, can we give the user of the Shrub class the appearance of dealing with a tree that does not actually exist?

We have already taken the first step towards doing that: the actual data structure is completely hidden by the Shrub and Tag classes. Although a user of those classes may think that a Shrub is really a tree and a Tag is really a pointer, the encapsulation permits us to implement it differently.

Suppose, for example, that we use the Shrub class to store just the preorder traversal of the tree in array form. Thus, for example, our representation of  $3+4$  is just the sequence  $(+, 3, 4)$  stored as a linear array:



The size is now down to three words aside from the dictionary. We might even be able to do better if we can store dictionary indices in units smaller than whole words. Even with the dictionary, the total size is now eight words, as opposed to ten in our first example, which is a significant gain even under the unrealistically pessimistic assumption that the dictionary isn't useful for anything else.

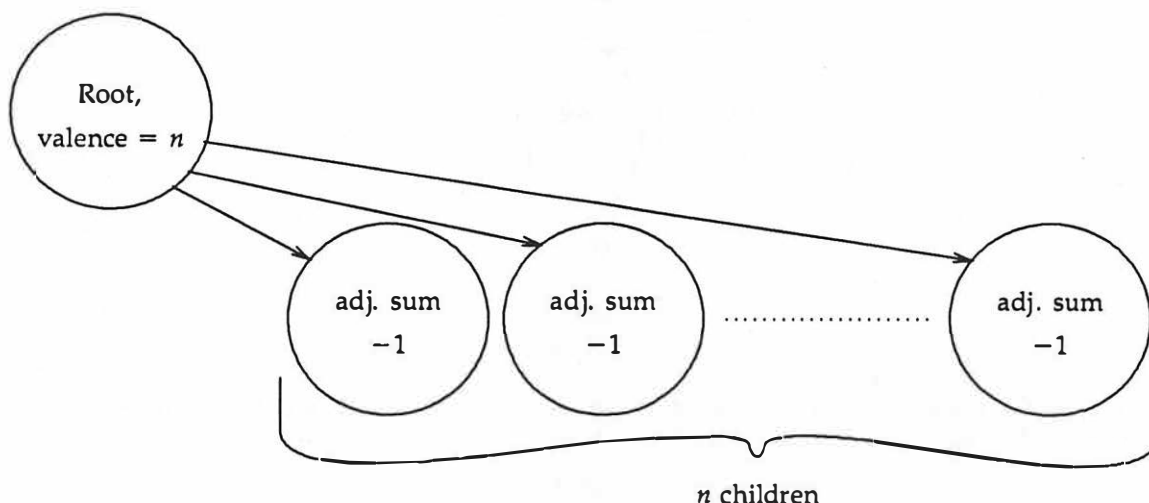
The ability to make this look to the user like a real tree depends on the ability to implement the Tag operations using only the preorder form. Intuitively it seems like this should be possible, and indeed, that intuition is confirmed as part of the implications of the following two theorems.

First, define the *adjusted sum* of a sequence of nodes as the sum of the number of children of each node minus the number of nodes. Thus, for example, the adjusted sum of the sequence  $(+, 3, 4)$  is  $-1$  because  $+$  has two children,  $3$  and  $4$  have no children, and there are three nodes overall. The adjusted sum of the sequence  $(+, 3)$  would be zero.

Note that by this definition the adjusted sum of a concatenation of sequences is the sum of the adjusted sums of the individual sequences. This fact makes it easy to compute the adjusted sum of any sequence: just treat the sequence as a concatenation of individual elements. In other words, start a counter at zero, and then for each element add its valence and subtract one. The following theorem proves that we can start at the beginning of a subtree, compute the adjusted sum as we go along, and come up with  $-1$  at the end of the subtree.

**Theorem 1.** *The adjusted sum of a sequence that represents a complete (sub-)tree is  $-1$ . The proof is by induction on the height of the tree. If the height is 1, the tree must be a leaf and the theorem is immediately true by inspection. Suppose then that the height is  $n$  and that the root of our tree has  $k$  subtrees. Each subtree has height less than  $n$ , so by the induction hypothesis the adjusted sum of each subtree is  $-1$ . The adjusted sum of the whole tree is therefore  $-1$  times the number of subtrees plus the number of children the root has, minus 1 for the root itself. Since the number of subtrees is just the number of children of the root, the adjusted sum of the entire tree must be  $-1$ .*

More intuitively, a non-trivial (sub-)tree looks something like this:



This tree is deliberately drawn lopsided to suggest the sequence in which its elements are stored in the corresponding Shrub array. It should be easy to see here that the adjusted sum of the root is  $n-1$  and the adjusted sum of the subtree rooted in each child is  $-1$  (by the induction hypothesis), so that the adjusted sum of the entire tree is  $-1$ .

This is not quite enough to prove that we can locate the end of a subtree without uncompressing it. Although we have proven that the adjusted sum will be  $-1$  at the end of the tree, we might also have hit  $-1$  somewhere along the way. The following theorem proves that that can't happen.

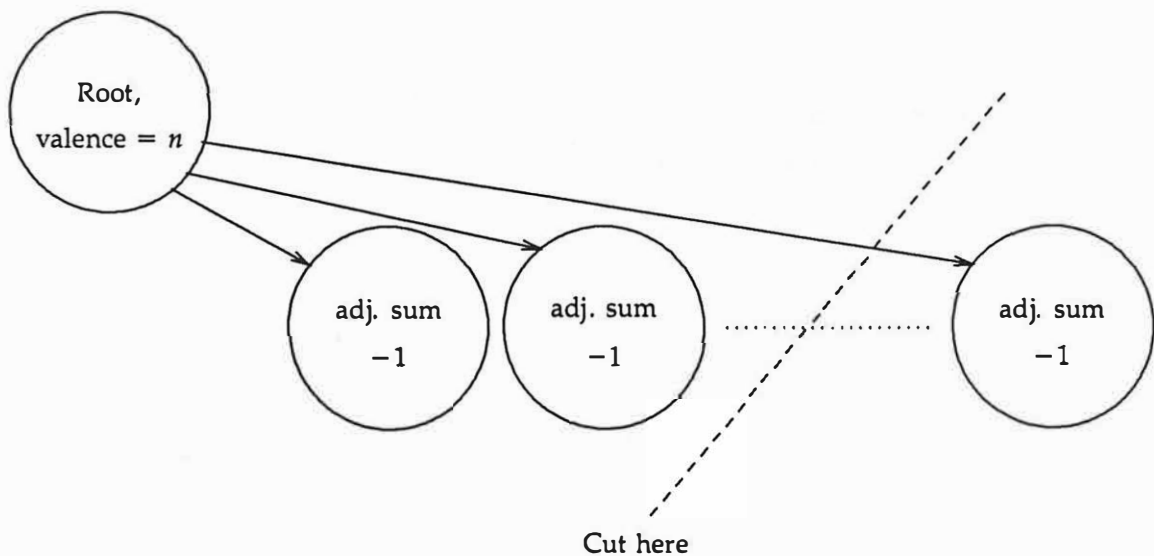
**Theorem 2.** *Suppose we have a nonempty sequence that represents a complete tree and we cut it into two subsequences with the right one nonempty. Then the adjusted sum of the left subsequence is  $\geq 0$  and the adjusted sum of the right subsequence is  $< 0$ . We note first that it suffices to prove that the adjusted sum of the right subsequence is negative because by Theorem 1 the adjusted sum of the entire sequence is  $-1$ . If the adjusted sum of the right subsequence is negative, the adjusted sum of the left subsequence must therefore be positive or zero.*

We can now prove the theorem by induction on the number of elements in the sequence. If the sequence has only one element, the requirement that the right subsequence be nonempty means that the left subsequence must be empty. That means that the right subsequence is the whole sequence, which therefore has an adjusted sum of  $-1$ .

What if the sequence has more than one element? If the left subsequence is empty, the right subsequence is again the whole sequence and we're done. Assume therefore that neither subsequence is empty.

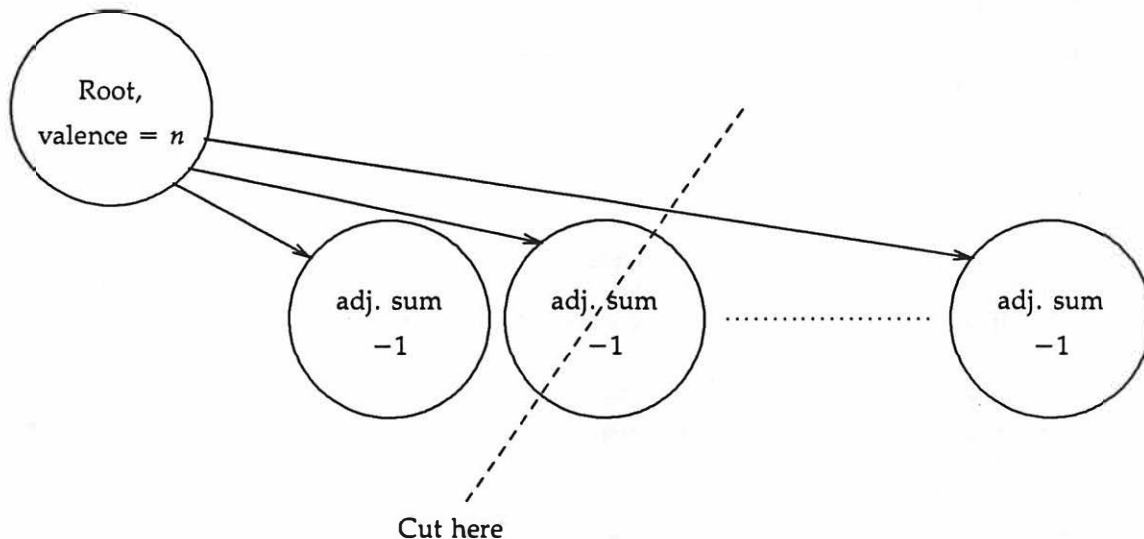
What does the sequence look like? Because it's in preorder, it is a root followed by one or more subtrees. The root is in the left subsequence because the left subsequence is nonempty, which means that the cut is either between two of the subtrees (or between the root and the first subtree) or in the middle of a subtree.

If the cut is between two subtrees, the right subsequence is a sequence of one or more complete subtrees, each of which has an adjusted sum of  $-1$ , so we're done. Intuitively, the situation looks like this:



Because there is one or more complete subtrees on the right of the cut, the adjusted sum of the right-hand part must be negative.

That leaves the case where the cut is in the middle of a subtree:



Note that the subtree must have at least one node on each side of the cut; otherwise the previous case covered it. Look at the subtree that was cut. By the induction hypothesis, the adjusted sum of the part of the subtree to the right of the cut is negative. After the partial subtree comes zero or more complete subtrees, each of which has an adjusted sum of  $-1$ . Thus the adjusted sum of the right subsequence is the sum of a negative number and zero or more other negative numbers, which is negative.

These two theorems prove a very important property: *if you accumulate the adjusted sum of a sequence while scanning from left to right, that sum will be negative if and only if you have scanned a complete subtree, at which point it will be  $-1$ .* Theorem 1 proves the 'if' part of this assertion and Theorem 2 proves the 'only if' part. That means, for example, that if  $p$  points at some element in a Shrub, and we know that element has a right sibling, we can cause  $p$  to point at that sibling as follows:

```

int count = 0;
do count += valence(*p++) - 1;
while (count >= 0);

```

Recall that the user supplies the `valence` function, which returns the number of children of a node given to it as argument.

Similarly straightforward algorithms can be found to do other interesting things. For example, here is the `move_down` operation:

```

if (n > 0 && n < valence(*p)) {
    // We know that p points to a node
    // with at least n children.

    // First we make p point at the first child,
    p++;

    // ... and then we move p to its right sibling n-1 times
    for (int i = 1; i < n; i++) {
        int count = 0;
        do count += valence(*p++) - 1;
        while (count >= 0);
    }
}

```

and here is the `move_up` operation:

```
int sum = 0;
do sum += valence(*--p) - 1;
while (sum < 0);
```

## Experience

It is clear that this data structure can eliminate pointers completely, thus halving memory consumption under our earlier assumptions. In fact, it may be possible to do even better than that if we can find a way to store dictionary indices that takes less than a full word each.

What about execution time? It is clear that many operations that can be done in  $O(1)$  time with conventional data structures now require  $O(n)$ , where  $n$  is the number of elements in the relevant subtree. However, finding the parent of a node is actually faster than in a conventional tree, unless the tree explicitly stores pointers from nodes to their parents.

Moreover, one common operation is looking at every element of a (sub-)tree in preorder. This operation is  $O(n)$  in both representations. However, the Shrub data structure is simpler to traverse, so its traversal should be a constant factor faster.

For that matter, visiting the nodes in postorder is no harder with a Shrub than it is for a conventional tree representation. Briefly, it involves keeping a stack that represents the path from the root to the current node. Each stack item includes a count of the children of that node that have yet to be visited. Each leaf decrements the count on top of the stack; when it reaches zero, you visit the root of the subtree on top of the stack and then pop the stack.

For the kinds of applications for which Shrubs are useful, we believe that these large-scale traversal and search operations will dominate applications' execution time, so the fact that other operations are slower may not be relevant at all.

For example, consider again our hypothetical browser. Searching a Shrub for references to a particular variable should be at least as fast as for a conventional tree. Moreover, once we have found a node, we will often want to display the surrounding context. If that context is a complete subtree, such as a function body, then the nodes to display will be the ones that will be stored near the one we've found, which means that the display operation will also be reasonably fast. The slower operations, such as moving from a node to an arbitrary sibling, are likely to be required only in response to input from the user. In that case, the slower execution is likely to go unnoticed.

Moreover, the fact that Shrubs are pointer-free means that it is likely to be easier to store them for later retrieval. If the items in the Shrub are dictionary indices, it's just a matter of storing the dictionary (which would presumably be necessary anyway) and then simply blasting the bits of the Shrub out onto disk.

We have been using Shrubs for compact storage of data structures for our C++ development environment kernel. In this case, the objects involved carry more overhead than just the pointers necessary for a conventional tree structure: in addition to back pointers, there are pointers associated with C++ virtual tables and virtual base classes. It is too early to know for certain, but our present estimates are that careful use of Shrubs reduces our memory requirements by at least a factor of eight.

## Acknowledgments

Thanks to Steve Buroff, Martin Carroll, Barbara Moo, Rob Murray, and Bjarne Stroustrup for their careful reading of drafts of this paper and many comments.



## References

In *A Statically Typed Abstract Representation for C++ Programs* (Proceedings 1992 Usenix C++ Conference), Rob Murray describes Alf, which is the part of Grail that uses this work.

In *Fundamental Algorithms* (section 2.3.3), D. E. Knuth mentions the idea of storing a tree as a (postorder) traversal along with the number of children of each node, along with some related algorithms.

In *Graph Theory and Computing* R. C. Read discusses algorithms for encoding trees in compressed forms, although not for traversing trees without first decompressing them. The algorithm described here is essentially what Read calls *walk-around valency* coding; he proves theorems equivalent to the two independently proved in this paper.

An example of more traditional tree storage appears in my column *An Example of Dynamic Binding in C++* in the *Journal of Object-Oriented Programming* (Volume 1, number 3, August/September 1988).

Bjarne Stroustrup is responsible for the observation that trends in computer hardware mean that saving space in data structure representations may save time as well. That observation is what led me to look for such a representation in the first place. He has described this and other notions related to the present work in several internal memoranda not yet published.



# High-Performance Scientific Computing Using C++

K. G. Budge, J. S. Peery and A. C. Robinson

*Computational Physics Research and Development (1431)*

*Sandia National Laboratories*

*Albuquerque, NM 87185-5800*

*kgbudge@sandia.gov, jspeery@cs.sandia.gov, acrobin@cs.sandia.gov*

## Abstract

Concepts from mathematics and physics often map well to object-oriented software since the original concepts are of an abstract nature. We describe our experiences with developing high-performance shock-wave physics simulation codes in C++ and discuss the software engineering issues which we have encountered. The primary enabling technology in C++ for allowing us to share software between our development groups is operator overloading for a number of “numeric” objects. Unfortunately, this enabling feature also impacts the efficiency of our computations. We describe the techniques we have utilized for minimizing this difficulty.

## Introduction

Developers of scientific software systems are tasked to implement abstract ideas and concepts. The software implementation of algorithms and ideas from physics, mechanics and mathematics should in principle be complementary to the mathematical abstractions. Often these ideas are very naturally implemented in an object-oriented style. For example, our group is developing software to solve the equations

$$\rho \frac{d^2 \vec{u}}{dt^2} = \nabla \cdot \mathbf{T} + \rho \vec{b} \quad (1)$$

$$\frac{d}{dt} \mathbf{T} = f\left(\nabla \frac{d\vec{u}}{dt}, \mathbf{T}, \dots\right) \quad (2)$$

subject to a variety of boundary conditions. These equations describe the mechanical response of a continuous medium. They contain numerous abstract mathematical objects such as scalar, vector, and tensor fields, arithmetic operators, and calculus operators. Physical concepts such as the equation of state of a physical substance and its constitutive response model can also be encapsulated very cleanly as an object. Similar objects are found in the equations describing many other physical systems. All of these can be represented by appropriate data structures in a high-level computer language.

Traditionally, scientists have relied on FORTRAN for high-performance computing. The reasons for this are clear. FORTRAN optimizing compilers have been around for a long time and produce extremely efficient executable code. Large libraries of numerical routines are available in the language. It is familiar to almost every scientific worker. The drawbacks of standard FORTRAN are also well-known. For example, it is devoid of mandated type checking, does not support structures and the concept of a free memory store does not exist. Note that these statements are true of *standard* FORTRAN-77. Some vendors have provided extensions to the language in order to remedy some of these deficiencies. For example, most vendors now support the nonstandard DO-ENDDO construct. Also, codes using large databases and complicated algorithms *must* find a way to dynamically allocate memory. Since no provisions for this exists in standard FORTRAN-77, programmers must either use proprietary extensions which supply the existing functionality, make calls to

system level routines based on C or develop intricate memory management schemes which access common memory.

It is not unknown for a large scientific production code written in FORTRAN to exceed half a million lines in length and to include numerous platform-dependent statements. This represents a maintenance challenge comparable to that for a small operating system. Furthermore, such code tends to be very opaque. Unless the code is extremely well-documented, the transfer or retirement of one of its programmers can effectively freeze portions of the code.

Many scientific programming groups have come to the realization that such programming practices are prohibitively expensive. What is needed is a programming environment in which code is highly reusable, transparent to the reader, and easily debugged during development and maintenance, but which retains excellent efficiency. As a result, there is a growing interest in more modern languages in the scientific community. C++ has attracted the most interest because of its wide availability and support in the general programming community and because of its explicit design goals to provide object-oriented functionality and excellent software engineering characteristics in a way which also supports efficient code execution. Applications utilizing C++ include distributed particle simulations, partial differential equations, fluid mechanics, robotic languages, mesh generation, adaptive grid methods, data-parallel C, general numeric libraries, image algebras, large scientific database management and genetic algorithms. See references [1]-[20] which are organized roughly in chronological order.

It should be pointed out that a Fortran 90 ANSI standard has recently been adopted [21]. In addition a specifically non-ANSI based effort is underway to agree upon a High Performance Fortran standard this year. The final universal acceptance and wide availability of compilers for these languages is likely, but on an uncertain time scale. A major reason why we chose not to do our current development work in a Fortran 90 style language was the unclear state of the standard at the time we began our work as well as very uncertain availability and support considerations on our rapidly changing target architectures. In addition, current language specifications do not appear to provide some of the robust software engineering characteristics which we have come to enjoy in C++.

In this paper we briefly describe our use of C++ in two large-scale scientific code development projects and give examples of how we have mapped physical, mathematical and computational concepts to C++. One project is developing a shock wave physics simulation code (PCTH) for fixed "Eulerian" grids with massively parallel MIMD architectures as the primary target, and the other project (RHAE++) uses an "Arbitrary Lagrangian-Eulerian" technology based on an unstructured grid finite element technology. In the following sections we describe several object-oriented concepts which we are utilizing and how we are attempting to realize the power of the abstraction capability and the excellent software engineering features which the C++ language provides. Subsequently, we discuss some aspects of the efficiency difficulties which we have encountered as well as approaches to their resolution.

## C++ As a Meta-Language for Mathematical Physics

We tend to regard C++ as being essentially a *meta-language* whose dialects can be tailored to a particular field of application. Our dialects of C++ are tailored to mathematical physics. For example, the polar decomposition of a velocity gradient  $\mathbf{L}$  is expressed by the equations [22]

$$\mathbf{D} = \frac{1}{2} (\mathbf{L} + \mathbf{L}^T) = \text{Sym}(\mathbf{L}) \quad (3)$$

$$\mathbf{W} = \frac{1}{2} (\mathbf{L} - \mathbf{L}^T) = \text{Anti}(\mathbf{L}) \quad (4)$$

$$\dot{\mathbf{z}} = [\epsilon_{ijk} V_{jm} D_{mk}] = \text{Dual}(\mathbf{V} \mathbf{D}) \quad (5)$$

$$\vec{\omega} = \text{Dual}(\mathbf{W}) - 2 (\mathbf{V} - \text{Tr}(\mathbf{V})\mathbf{1})^{-1} \dot{\mathbf{z}} \quad (6)$$

$$\Omega = \frac{1}{2} \text{Dual}(\vec{\omega}) \quad (7)$$

$$\frac{d\mathbf{R}}{dt} = \Omega \mathbf{R} \quad (8)$$

$$\frac{d\mathbf{V}}{dt} = \mathbf{L}\mathbf{V} - \mathbf{V}\Omega \quad (9)$$

The time discretization used to integrate the last two equations is

$$\mathbf{R}^{n+1} = \left(1 - \frac{1}{2}(\Delta t)\Omega\right)^{-1} \left(1 + \frac{1}{2}(\Delta t)\Omega\right) \mathbf{R}^n \quad (10)$$

$$\mathbf{V}^{n+1} = \mathbf{V}^n + \text{Sym}(\mathbf{L}\mathbf{V} - \mathbf{V}\Omega)\Delta t \quad (11)$$

In RHALE++ we have defined classes representing the vector and tensor objects in these equations [23]. Using these classes, we can code this algorithm as

```
void Decompose(const double delt, SymTensor& V,
               Tensor& R, const Tensor& L)
{
    SymTensor D;
    AntiTensor W, Omega;
    Vector z, omega;

    D = Sym(L);
    W = Anti(L);

    z = Dual(V*D);
    omega = Dual(W) - 2.0 * Inverse(V - Tr(V) * One) * z;
    Omega = 0.5 * Dual(omega);

    R = Inverse(One - 0.5 * delt * Omega) *
        (One + 0.5 * delt * Omega) * R;
    V += delt * Sym(L * V - V * Omega);
}
```

Note the heavy use of operator overloading. This code is transparent and its underlying class libraries are versatile and easy to maintain. A physicist familiar with the polar decomposition algorithm can make immediate sense of this code fragment without the need for any additional documentation.

By contrast, the FORTRAN version of this subroutine is

```
subroutine decompose(delt,V_xx,V_xy,V_xz, V_yy,
* V_yz, V_zz, R_xx, R_xy, R_xz, R_yx, R_yy, R_yz,
* R_zx, R_zy, R_zz, L_xx, L_xy, L_xz, L_yx, L_yy,
* L_yz, L_zx, L_zy, L_zz)
    D_xx = L_xx
    D_xy = 0.5*(L_xy+L_yx)

... about three pages of these proceedings. . .

V_xy = V_xy + 0.5*(t3_xy + t3_yx)
V_zz = V_zz + t3_zz
return
end
```

The stylistic advantages of C++ are obvious. The second subroutine (in FORTRAN) is virtually unreadable. It is also very difficult to debug. However, the FORTRAN version is somewhat more efficient. Several expressions are evaluated in the C++ version that are never used. In principle, a sufficiently intelligent optimizer could eliminate these expressions.

Field classes representing scalar, vector, and tensor fields are fundamental to our approach to simulation coding. Field classes are coded as sets of smart arrays representing the components of vector or tensor fields. At this level, no topological information is included in the fields. Thus, while numerous element-by-element operations are overloaded, no calculus operations are defined. These are added in classes derived from the basic field classes.

These field classes hide subscripting and loops, eliminating a common source of error in FORTRAN code. In the FORTRAN code fragment above, the functional interface looks essentially the same whether the arguments are scalars or data-parallel arrays. In the C++ case the same holds true except the compactness of the high-order tensor fields hides the extent of the implied data and the interactions between tensor elements implied by the mathematical operations.

## Encapsulation of Physical Concepts

In our simulation codes empirical equations are used along with the basic conservation equations to describe how a given material behaves. In shock physics, two of these empirical equations are an equation of state and a constitutive model. An equation of state generally determines the pressure, temperature, and sound speed of a material based on the material's density and energy. A constitutive model, on the other hand, determines the stress state of a material based on the material deformation. Within each of these concepts numerous models are available; however, the input and the output quantities which any equation of state utilizes and provides are the same. C++ can be used to encapsulate the uniqueness of a model (particularly the private data) and provide a common interface to the concept.

Using C++ to encapsulate physical concepts can be best seen in an equation of state class. Encapsulation begins with the concept of an abstract class that contains all data and functions common to every equation of state. A highly simplified summary of such a class is given as

```
class Equation_of_State {
public:
    Equation_of_State();
    Equation_of_State(const Equation_of_State&);
    ~Equation_of_State();
    virtual void Update_Thermodynamic_State(
        const Field& density,
        const Field& energy,
        Field& pressure,
        Field& temperature,
        Field& sound_speed) const = 0;
};
```

This abstract class provides a common interface to updating the thermodynamic state of materials. Constructing a particular equation of state simply requires deriving a new class from the abstract class, adding any unique data, and providing a unique function for updating the thermodynamic state of the material. For example, an ideal gas equation of state class is given as

```
class Ideal_Gas : public Equation_of_State {
private:
    double gamma; // Ratio of specific heats
    double cv; // Specific heat
    double gamma_minus_one; // gamma - 1
public:
    Ideal_Gas();
    Ideal_Gas(const Ideal_Gas&);
```

```

~Ideal_Gas();
void Update_Thermodynamic_State(
    const Field& density,
    const Field& energy,
    Field& pressure,
    Field& temperature,
    Field& sound_speed) const;
};

```

The third private variable ( $\gamma-1$ ) is indicative of the internal constants which a given equation of state might create in order to more efficiently perform its functions as a server class for fields which are passed to it. To add an additional equation of state option to the physics code one only has to modify the code at the one location where a particular type of equation of state is specified for a given material. The rest of the code utilizes only the base class pointer and the correct functions are called at run time. In addition, by careful design, a class that abstracts the physics can be used by many codes (the equation of state and constitutive classes are being designed to be used both by PCTH and RHALE++) and can be extended quickly. For example, it has taken less than 4 hours to add a complicated new constitutive model to the RHALE++ code.

## Tracer Particle Objects

PCTH is being developed with massively parallel MIMD architectures as a primary focus although with a data-parallel syntax to facilitate access to vector hardware. One of the requirements for this code is for something that is called a *tracer particle*. These are points which do not interact with the simulation flow field but are required to follow the flow. Since the PCTH code decomposes work spatially by subdividing the simulation space, these particles (objects) need to skip from node to node of the massively parallel machine according to the dynamics of the flow field. These particles store an identifier, position, velocity and local state values. They know how to send themselves to neighboring nodes as well as how to create a copy of a particle which has been sent from a neighboring node. The physics code knows how to interpolate state values onto the tracer particles.

## C++ Software Engineering Experience

At this point in time there are relatively few classes for purchase or in the public domain that address the needs of scientific community and, of these, none have been ported and optimized for vector and parallel computing architectures. In addition, there are no classes for scientific computing that have been adopted as a standard. Part of the philosophy behind C++ is not to “re-invent the wheel;” however, at this time, most of the burden of developing base classes, such as matrix, array, and vector classes, falls on the members of our project teams. As a result, there is a proliferation of scientific class libraries, each with their own particular syntax, functionality, and performance.

The redundancy of these scientific classes is very unfortunate because developing robust base classes is extremely expensive. One must hope that the lifetime of the code project is long enough to reap the rewards of the initial investment in developing the base classes.

The RHALE++ and PCTH projects both began with their own flavors of Field classes. The RHALE++ project implemented the field classes as arrays of objects while the PCTH project implemented an object of arrays approach. However, it was realized that although the initial investment of developing a robust array classes is expensive, it is equally expensive to port and tune the class to a particular computer architecture. As a result, a common array class, which serves as the building block for scalar, vector, and tensor fields, is being developed as the basis for performing mathematical operations in both codes in an “object of arrays” approach. We feel that this is a wise investment since the performance of the array class is a direct measure of the performance of these two codes and thus only this class must be tuned for a particular computer architecture and both codes will benefit.

Although developing robust base classes is costly, extending or deriving from these classes is relatively inexpensive. For example, adding an additional operation between a vector and tensor class is very inexpen-

sive since the array class components of each can be operated upon as single objects.

A major reason for going with C++ was the complexity of the algorithms which we were dealing with in the case of the RHALE++ project and the complexity of the underlying (current and unknown future) parallel architectures in the case of the PCTH project. By using object-oriented concepts we can hide the underlying architectures to a great degree and thus enable us to quickly port to any machine on which we may be required to run.

The fact that C++ is based on C gives us confidence that a C++ environment can be provided relatively cheaply on any architecture of interest. These architectures potentially include a variety of MIMD, SIMD (data-parallel) and vector processor machines as well as networks of high performance workstations. In addition we hope to eventually take full advantage of the workstation software tools market even on the exotic architectures with which we must deal. We believe that much of this software will be written in C++ in the future. In fact, in PCTH we are already utilizing a commercial solid modeling toolkit written in C++ to implement the volume fraction computations in the initialization phase of our simulations. As we consider balancing production software development with the uncertainties of the future, C++ has appeared to be an excellent bet. Our rationale is similar to common arguments given for moving to C++ for production development [24].

One of the most enticing aspects of C++ is the elimination of many common bugs introduced in Fortran coding. Many of these bugs are removed by the strong type checking and function argument matching capabilities inherent in C++. With the use of overloaded operators, index mistakes are eliminated or reduced to a single location in the code which can be easily identified and corrected. For most scientific applications, ninety percent or more of the bugs have just been eliminated. However, for most Fortran programmers that switch to C or C++, a new and much more nasty bug is potentially introduced with the concept of pointers. Fortunately, very nice interactive C++ debugging environments have been developed to support the large C++ programming community.

In the RHALE++ and PCTH projects, it has been our experience that we spend much less time debugging executable C++ code than FORTRAN code. Most of our debugging activities center on correcting the physics algorithms (incorrectly coded or poor selection of algorithms) and sometimes in locating bad pointers. This excellent experience is due mainly to our use of overloaded operators and the robust error detection features of C++ compilers.

## C++ Reusability

Another aspect of C++ and object-oriented programming is the ease with which new physical models can be added to a standing code and ported between platforms and projects. Both the PCTH and RHALE++ projects are developing large and complex codes which will be evolving for a number of years. PCTH uses a finite-difference method, whereas RHALE++ uses a finite-element method. These two approaches to numerical calculus differ considerably.

Despite the profound difference in overall methods, we are now in the process of merging code portions which are conceptually identical. This is possible because of the field class concept which is inherent in the underlying physics and thus the codes themselves. We are now converting the two codes to use the same set of base field classes, which are then specialized for finite difference/finite element field classes. Both codes must eventually run on a wide variety of platforms, ranging from PCs to the newest massively parallel computers. The architectural differences between platforms require different optimization strategies. We are confident that these can be hidden to a great degree in the field class libraries. The need to write efficient code for massively parallel computers and the prospect of hiding domain decomposition and message passing in the field class libraries was a strong incentive to investigate scientific C++ in the first place.

The equation of state and strength models are derived from abstract base classes but take fields as arguments. It does not really matter whether the underlying field is based on field elements or finite differences. This makes it relatively simple to incorporate new equations of state or strength models in the codes.

The original equation of state code was developed for PCTH. It was then moved in a straightforward way to RHALE++ and the new upgraded capabilities will move readily back to PCTH. At this time we are converg-



ing on the correct specification and approach.

Complicated algorithms can also benefit from abstract formulations. We have had the experience of moving a complicated interface tracking algorithm from PCTH to RHALE++ as a single block of code with only very minor modifications.

The fundamental reason why we are able to move in the direction of better code reuse is because of the common underlying physical concepts which must be modeled by our two codes. The connecting link is the use of field classes which can be considered to be a "numeric" types with appropriate overloaded operators. Unfortunately, the overloaded operator feature of C++, which enables these excellent maintainability, reusability and extensibility characteristics for our development projects, is also the major impediment for efficient execution of our codes. These difficulties and the extent to which they have been solved will now be described.

## C++ Efficiency Issues for Overloaded Operators

The advantages of C++ described above are unfortunately offset to some degree by the difficulty of developing efficient overloaded operator methods for objects representing large arrays. Many high performance machines require array (vector) operations to achieve good performance. The fact that the current C++ language and implementations do not support compiler optimization of large arithmetic objects, as if they were fundamental types, causes extensive difficulties for developing efficient C++ applications.

This is illustrated in Table 1, which shows the estimated processor speeds for test versions of the polar decomposition algorithm. Both C++ and FORTRAN-77 versions were tested on Sun workstations and Sandia's CRAY-YMP for a range of field sizes. The C++ version uses reference counting and memory management techniques discussed later in this section. In addition, innermost loops are implemented through calls to FORTRAN subroutines (using the C++ external linkage specification facility). The statistics show that peak speed for pure FORTRAN-77 is two or three times greater than that for C++ and that C++ involves a considerable overhead (reflected in the array size at which one gets 50% of peak performance).

**Table 1. Performance of Field Classes In a Polar Decomposition Test Problem**

# elements	Sun Sparc-2 MFlops	Sun Sparc-2/F MFlops	Cray YMP MFlops	Cray YMP/F MFlops
1	0.04	--	0.11	12.7
10	0.31	1.3	1.03	63.7
100	0.92	3.1	9.6	185.8
1000	0.89	2.7	59.1	221.9
10000	0.88	2.7	123.7	225.4
100000	out of memory	out of memory	140.0	225.9
50% peak	18 elements	13 elements	1380 elements	27 elements

These figures are somewhat discouraging taken at face value although they are consistent with previously published results [1][4][15]. The Cray is a vector machine and its overall performance is extremely sensitive to scalar portions of the code. The scalar overhead tends to be very large except when vector lengths are large. The peak speed of C++ is about half of the Fortran speed. This is due to the fact that the CRAY architecture is able to chain (i.e. perform certain multiply-add operations in parallel).

What accounts for these results? The problem is that C++ strongly isolates binary operations. Consider the

following test program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

class Array {
public:
    Array(void);
    Array(const int n);
    Array(const int n, const double[]);
    Array(const Array&);
    ~Array(void);
    Array& operator=(const Array&);

    double& operator[](const int);

    friend Array operator*(const Array&, const Array&);
    friend Array operator+(const Array&, const Array&);

private:
    int n;
    double *data;
};

inline Array::Array(void) :
    n(0),
    data(NULL)
{}

inline Array::Array(const int nn) :
    n(nn),
    data(new double[nn])
{}

inline Array::Array(const int nn, const double d[]) :
    n(nn),
    data(new double[nn])
{
    memcpy(data, d, sizeof(double)*n);
}

inline Array::Array(const Array& src) :
    n(src.n),
    data(new double[src.n])
{
    memcpy(data, src.data, sizeof(double)*n);
}

inline Array::~~Array(void) {
    delete data;
}

inline Array& Array::operator=(const Array& src) {
    delete data;
    n = src.n;
    data = new double[n];
```

```

    memcpy(data, src.data, sizeof(double)*n);
    return *this;
}

inline double& Array::operator[](const int nn){
    if (nn<0 || nn > n) abort();
    return data[nn];
}

inline Array operator*(const Array& a, const Array& b){
    Array result(a.n);
    for (register i=0; i<a.n; i++)
        result.data[i] = a.data[i] * b.data[i];
    return result;
}

inline Array operator+(const Array& a, const Array& b){
    Array result(a.n);
    for (register i=0; i<a.n; i++)
        result.data[i] = a.data[i] + b.data[i];
    return result;
}

main(){
    const int SIZE = 50000;
    static double a_data[SIZE] = { 2., 5., 3., 8. };
    static double b_data[SIZE] = { 5., 3., 4., 2. };
    static double c_data[SIZE] = { 3., 4., 5., 6. };
    static double x_data[SIZE] = { 2., 5., 7., 9. };

    Array A(SIZE, a_data);
    Array B(SIZE, b_data);
    Array C(SIZE, c_data);
    Array X(SIZE, x_data);
    Array Y;

    for (register i=0; i<100; i++){
        Y = C + X*(B + X*A);
    }

    printf("First element is %f\n", Y[0]);
}

```

The C and FORTRAN equivalents of this program achieve a peak performance of 230 MFlops on the CRAY-YMP. This unoptimized C++ version achieves only 49 MFlops.

The CFRONT translator generates code which is essentially equivalent to

```

struct Array {
    int n ;
    double *data ;
};

static char multiply_Array_Array(
    struct Array *result,
    struct Array *a,
    struct Array *b )

```

```

{
    struct Array Result ;
    register int i ;
    int itmp ;

    itmp = a->n;
    Result.n = itmp;
    Result.data = (double*)malloc(sizeof(double) * itmp);

    for(i=0;i < a->n ;i ++ )
        Result.data[i] = a->data [i] * b->data [i];

    result->n = Result.n;
    result->data =
(double*)malloc(sizeof(double)*Result.n);
    memcpy(result->data, Result.data,
sizeof(double) * result->n);

    free (Result.data);
    return ;
}

static char add_Array_Array (
    struct Array *result,
    struct Array *a,
    struct Array *b )
{
    struct Array Result ;
    register int i ;
    int itmp ;

    itmp = a->n;
    Result.n = itmp;
    Result.data = (double *)malloc(sizeof(double)*itmp);

    for(i=0;i < a->n ;i ++ )
        Result.data[i] = a->data[i] + b->data[i];

    result->n = Result.n;
    result->data =
(double *)malloc(sizeof(double)* Result.n);
    memcpy(result->data, Result.data ,
sizeof(double)* result->n);

    free (Result.data );
    return ;
}

int main () {

    static double a_data [50000]= { 2. , 5. , 3. , 8. } ;
    static double b_data [50000]= { 5. , 3. , 4. , 2. } ;
    static double c_data [50000]= { 3. , 4. , 5. , 6. } ;
    static double x_data [50000]= { 2. , 5. , 7. , 9. } ;

    struct Array A ;
    struct Array B ;

```

```

struct Array C ;
struct Array X ;
struct Array Y ;

register int i ;

A.n = 50000;
A.data = (double *)malloc (sizeof(double)* 50000);
memcpy(A.data , (double*)a_data, sizeof(double)* A.n);

B.n = 50000;
B.data = (double *)malloc (sizeof(double)* 50000);
memcpy(B.data , (double*)b_data, sizeof(double)* B.n);

C.n = 50000;
C.data = (double *)malloc (sizeof(double)* 50000);
memcpy(C.data , (double*)c_data, sizeof(double)* C.n);

X.n = 50000;
X.data = (double *)malloc (sizeof(double)* 50000);
memcpy(X.data , (double*)x_data, sizeof(double)* X.n);

for(i=0;i < 100 ;i ++ ) {
    struct Array atmp1 ;
    struct Array atmp2 ;
    struct Array atmp3 ;
    struct Array atmp4 ;

    multiply_Array_Array(&atmp1, &X, &A);
    add_Array_Array(&atmp2, &B, &atmp1);
    multiply_Array_Array(&atmp3 , &X, &atmp2);
    add_Array_Array(&atmp4, &C, &atmp3);

    free(Y.data);
    Y.n = atmp4.n;
    Y.data = (double *)malloc(sizeof(double)* Y.n);
    memcpy(Y.data, atmp4.data, sizeof(double)* Y.n);

    free(atmp4.data);
    free(atmp3.data);
    free(atmp2.data);
    free(atmp1.data);
}

printf ("First element is %f\n", (0 > Y.n ? abort(),
0 : Y.data[0]));\

free(Y.data);
free(X.data);
free(C.data);
free(B.data);
free(A.data);
}

```

Loops are not inlined. Thus, although they vectorize individually, they cannot be chained. In addition, considerable effort is wasted allocating and de-allocating memory for the temporaries and copying the contents

of temporaries to local variables.

The latter difficulty is solvable through the well-known technique of *reference counting*. Using this technique, the definition of class `Array` is changed as follows:

```
class Array {
public:
    Array(void);
    Array(const int n);
    Array(const int n, const double[]);
    Array(const Array&);
    ~Array(void);
    Array& operator=(const Array&);

    double& operator[](const int);

    friend Array operator*(const Array&, const Array&);
    friend Array operator+(const Array&, const Array&);

private:
    int n;
    int *ref_count;
    double *data;
};

inline Array::Array(void) :
    n(0),
    data(NULL),
    ref_count(NULL)
{}

inline Array::Array(const int nn) :
    n(nn),
    ref_count(new int(1)),
    data(new double[nn])
{}

inline Array::Array(const int nn, const double d[]) :
    n(nn),
    ref_count(new int(1)),
    data(new double[nn])
{
    memcpy(data, d, sizeof(double)*n);
}

inline Array::Array(const Array& src) :
    n(src.n),
    data(src.data),
    ref_count(src.ref_count)
{
    if (ref_count) (*ref_count)++;
}

inline Array::~~Array(void) {
    if (ref_count &&!--*ref_count) {
        delete data;
        delete ref_count;
    }
}
```

```

    }
}

inline Array& Array::operator=(const Array& src) {
    if (ref_count && !--*ref_count) {
        delete ref_count;
        delete data;
    }
    n = src.n;
    ref_count = src.ref_count;
    if (ref_count) (*ref_count)++;
    data = src.data;
    return *this;
}

```

With this change, the peak computation rate jumps to 84 MFlops -- a 70% increase, but still far short of the 230 MFlops achieved with conventional C coding.

Profiling of our test code shows that, for moderate array sizes, much time is spent in the memory allocation and deallocation routines. Our experience is that the scalar overhead can be significantly reduced by taking charge of memory management through overloaded new and delete operators. If we use the memory manager we developed for the field class library, the computation rate jumps from 3.7 to 8.5 MFlops for small arrays (~128 elements).

Now consider some hypothetical future enhancements to the compiler. If loops inlined, the translator would produce code within the main program loop equivalent to

```

for(i=0; i < 100 ; i ++ ) {
    struct Array atmp1 ;
    struct Array atmp2 ;
    struct Array atmp3 ;
    struct Array atmp4 ;
    struct Array Result ;
    register int ii;

    Result.n = X.n;
    Result.data = (double *)malloc(sizeof(double)*X.n);
    Result.ref_count = (int *)malloc(sizeof(int));
    *Result.ref_count = 1;

    for(ii=0; ii < X.n; ii++)
        Result.data[ii] = X.data[ii] * A.data[ii];

    atmp1.n = Result.n;
    atmp1.data = Result.data;
    atmp1.ref_count = Result.ref_count;
    if (atmp1.ref_count) (*atmp1.ref_count)++;

    if (Result.ref_count && ! --(*Result.ref_count)) {
        free (Result.data);
        free (Result.ref_count);
    }

    Result.n = B.n;
    Result.data = (double *)malloc(sizeof(double)*B.n);
    Result.ref_count = (int *)malloc(sizeof(int));
    *Result.ref_count = 1;
}

```

```

for(ii=0; ii < B.n; ii++ )
    Result.data[ii] = B.data[ii] + atmp1.data[ii];

atmp2.n = Result.n;
atmp2.data = Result.data;
atmp2.ref_count = Result.ref_count;
if (atmp2.ref_count) (*atmp2.ref_count)++;

if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
}

Result.n = X.n;
Result.data = (double *)malloc(sizeof(double)*X.n);
Result.ref_count = (int *)malloc(sizeof(int));
*Result.ref_count = 1;

for(ii=0; ii < X.n; ii++ )
    Result.data[ii] = X.data[ii] * atmp2.data[ii];

atmp3.n = Result.n;
atmp3.data = Result.data;
atmp3.ref_count = Result.ref_count;
if (atmp3.ref_count) (*atmp3.ref_count)++;

if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
}

Result.n = C.n;
Result.data = (double *)malloc(sizeof(double)*C.n);
Result.ref_count = (int *)malloc(sizeof(int));
*Result.ref_count = 1;

for(ii=0; ii < C.n; ii++ )
    Result.data[ii] = C.data[ii] + atmp3.data[ii];

atmp4.n = Result.n;
atmp4.data = Result.data;
atmp4.ref_count = Result.ref_count;
if (atmp4.ref_count) (*atmp4.ref_count)++;

if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
}

if (Y.ref_count && ! --(*Y.ref_count)){
    free(Y.ref_count);
    free(Y.data);
}

Y.n = atmp4.n;
Y.ref_count = atmp4.ref_count;
if (Y.ref_count) (*Y.ref_count)++;

```



```

Y.data = atmp4.data;

if (atmp4.ref_count && !--(*atmp4.ref_count)){
    free(atmp4.ref_count);
    free(atmp4.data);
}
if (atmp3.ref_count && !--(*atmp3.ref_count)){
    free(atmp3.ref_count);
    free(atmp3.data);
}
if (atmp2.ref_count && !--(*atmp2.ref_count)){
    free(atmp2.ref_count);
    free(atmp2.data);
}
if (atmpl.ref_count && !--(*atmpl.ref_count)){
    free(atmpl.ref_count);
    free(atmpl.data);
}
}
}

```

The individual loops are separated by substantial sections of code. Close examination of these reveals that the code can be rearranged:

```

for(i=0; i < 100 ; i ++ ) {
    struct Array atmpl ;
    struct Array atmp2 ;
    struct Array atmp3 ;
    struct Array atmp4 ;
    struct Array Result ;
    register int ii;

    Result.n = X.n;
    Result.data = (double *)malloc(sizeof(double)*X.n);
    Result.ref_count = (int *)malloc(sizeof(int));
    *Result.ref_count = 1;

    atmpl.n = Result.n;
    atmpl.data = Result.data;
    atmpl.ref_count = Result.ref_count;
    if (atmpl.ref_count) (*atmpl.ref_count)++;

    if (Result.ref_count && ! --(*Result.ref_count)){
        free (Result.data);
        free (Result.ref_count);
    }

    Result.n = B.n;
    Result.data = (double *)malloc(sizeof(double)*B.n);
    Result.ref_count = (int *)malloc(sizeof(int));
    *Result.ref_count = 1;

    atmp2.n = Result.n;
    atmp2.data = Result.data;
    atmp2.ref_count = Result.ref_count;
    if (atmp2.ref_count) (*atmp2.ref_count)++;

    if (Result.ref_count && ! --(*Result.ref_count)){
        free (Result.data);
    }
}

```

```

    free (Result.ref_count);
}

Result.n = X.n;
Result.data = (double *)malloc(sizeof(double)*X.n);
Result.ref_count = (int *)malloc(sizeof(int));
*Result.ref_count = 1;

atmp3.n = Result.n;
atmp3.data = Result.data;
atmp3.ref_count = Result.ref_count;
if (atmp3.ref_count) (*atmp3.ref_count)++;

if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
}

Result.n = C.n;
Result.data = (double *)malloc(sizeof(double)*C.n);
Result.ref_count = (int *)malloc(sizeof(int));
*Result.ref_count = 1;

atmp4.n = Result.n;
atmp4.data = Result.data;
atmp4.ref_count = Result.ref_count;
if (atmp4.ref_count) (*atmp4.ref_count)++;

if (Result.ref_count && ! --(*Result.ref_count)){
    free (Result.data);
    free (Result.ref_count);
}

if (Y.ref_count && ! --(*Y.ref_count)){
    free(Y.ref_count);
    free(Y.data);
}
Y.n = atmp4.n;
Y.ref_count = atmp4.ref_count;
if (Y.ref_count) (*Y.ref_count)++;
Y.data = atmp4.data;

for(ii=0; ii < X.n; ii++ )
    atmp1.data[ii] = X.data[ii] * A.data[ii];

for(ii=0; ii < B.n; ii++ )
    atmp2.data[ii] = B.data[ii] + atmp1.data[ii];

for(ii=0; ii < X.n; ii++ )
    atmp3.data[ii] = X.data[ii] * atmp2.data[ii];

for(ii=0; ii < C.n; ii++ )
    atmp4.data[ii] = C.data[ii] + atmp3.data[ii];

if (atmp4.ref_count && ! --(*atmp4.ref_count)){
    free(atmp4.ref_count);
    free(atmp4.data);
}

```

```

    }
    if (atmp3.ref_count && !--(*atmp3.ref_count)) {
        free(atmp3.ref_count);
        free(atmp3.data);
    }
    if (atmp2.ref_count && !--(*atmp2.ref_count)) {
        free(atmp2.ref_count);
        free(atmp2.data);
    }
    if (atmp1.ref_count && !--(*atmp1.ref_count)) {
        free(atmp1.ref_count);
        free(atmp1.data);
    }
}

```

This puts the loops together and permits chaining. The only assumptions made in rearranging the code in this manner are that *the allocation/deallocation routines have no side effects* and that *the allocation routine returns a pointer to unaliased memory*. This is equivalent to requiring that the global operators `new` and `delete` have no side effects and no aliasing. Failure to obtain memory in the allocation routine should throw an exception rather than returning a null pointer. If permitted to make these assumptions, an extremely intelligent compiler might eliminate the allocation/deallocation operations entirely.

Several other techniques for improving vector performance are known, although we have not implemented them in our present codes. The technique of *deferred expression evaluation* eliminates nearly all large temporaries, but with a significant overhead cost. The technique consists of building a parse tree for each expression at run time, which is only evaluated when it is assigned to a variable or otherwise used [25]. Temporaries contain tree nodes rather than data and use a relatively small amount of memory. Furthermore, one can apply optimizations to the parse tree, although the run time overhead involved may be prohibitive unless the arrays are very large.

An equivalent effect (with greatly reduced overhead) could be obtained by extending the C++ language to permit overloading of entire parse trees. For example, the signature

```
Array& operator+=(Array&, const Array&, const Array&)
```

might correspond to

```
a = b + c;
```

while

```
Array operator+*(const Array&, const Array&, const Array&)
```

would correspond to

```
a + b*c;
```

The chief objection to this approach is its complexity, particularly if one attempts to extend it to arbitrarily complex parse trees.

Another approach would be to permit users to specify optimizations along with the definition of a class and its operations. For example, one could instruct the compiler to replace all expressions of the form

```
a = b + c*d;
```

with the expression

```
a = c*d, a += b;
```

which might be easier to optimize because no memory allocations take place between the evaluation of the two sub-expressions.

Although we have no direct experience to date, the advent of return value optimizing compilers is encouraging [26].

The simplest solution may be to standardize an array class. Since the array class name would become a

reserved identifier, vendors would be free to develop compilers that implement the array class as a built-in type.

Even with the difficulties described above, we have seen that our C++ simulation codes can perform reasonably well with present C++ language systems. In the case of PCTH running on the nCUBE-2 hypercube we have implemented vendor-supplied assembly language routines for many of the operations in the base field class methods (though not yet in the calculus type operator classes). Results to date indicate that for sufficiently large granularities no more than about a 50 percent loss will be sustained relative to FORTRAN. Further optimization should improve this estimate. The required field size is larger than the granularity required for good efficiencies due to message passing overhead but not so large as to be completely unrealistic for utilizing the machine effectively for our simulations. By using the embedded assembly language routines, better than FORTRAN performance by over a factor of two has been observed for a simple, large granularity field test problem running on the nCUBE-2 [12]. It is unlikely that we will be able to approach this speed in the PCTH application itself (by increasing the field size on each node) due to the number of fields required by the PCTH code and the limited memory available (per node) on the nCUBE machine. On 1 CPU of the Cray Y-MP, the PCTH code has achieved 90 percent of the original CTH (FORTRAN 77) code speed on a fairly complicated two-dimensional problem with a 250x250 field granularity. In this case as well, smaller problems suffer from scalar code overhead as would be expected from the results described earlier. The reason the numbers compare so well for the CRAY is the fact that every array operation in the field classes has been carefully optimized and the fundamental vector lengths turn out to be much larger in the PCTH code than in the CTH code (due to the field abstraction). It is also possible that there are not sufficient chaining operations in the CTH algorithms to significantly boost overall performance over the binary operation limit. These results, for a base field class whose methods implement reference counting, internal memory management and specialized routines, are sufficiently encouraging for us to believe that C++ can become an extremely effective language for scientific and engineering programming as better class libraries, language features, and optimizing compilers become generally available.

## Summary

The abstractions and software engineering properties of the C++ language have been found to be an excellent fit to large scale scientific software development for the strong shock-wave physics codes which we are developing. We have found that considerable effort must be expended in the design (and redesign) of base classes which are to be used in our codes. However, once these classes are developed, we find that excellent control over the development of additional code is obtained. We have demonstrated that code based on object-oriented ideas can be readily reusable (or shareable) by other developers. We plan to share our code between projects to an even greater extent in the future as current classes are redesigned. Good reusability and shareability for our application classes and algorithms tends to hinge on the extensive use of operator overloading for objects which are essentially "numeric" types. Unfortunately, this is also the most difficult aspect of the language to implement and still retain efficiencies which approach C or Fortran. The various techniques which we have implemented to improve operator overloading efficiencies allow us to approach but not exceed Fortran efficiencies on the high-performance architectures which we have investigated as long as our objects are of sufficiently large granularity. Our current efficiency estimates are within acceptable limits, but we consider that much more attention must be paid to issues of optimization of numeric types both from the standpoint of compiler optimization and the language specification. The rapid increase in interest for using C++ that we observe in the scientific computing community implies that an opportunity to gain a strong foothold in this market is available. We consider that the many advantages for software development obtained by C++ are worth the price today but that continued rapid development of the fundamental technology with respect to operator overloading of numeric object types is essential to compete effectively in the future.

## Acknowledgments

This work was performed at Sandia National Laboratories supported by the U. S. Department of Energy under contract number DE-AC04-76DP00789.

## References

- [1] I. G. Angus and W. T. Thompkins, "Data Storage, Concurrency, and Portability: An Object Oriented Approach to Fluid Mechanics," The Fourth Conference on Hypercubes, Concurrent Computers and Applications, 1989.
- [2] R. J. Collins, "CM++: A C++ Interface to the Connection Machine," Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications, Marist College, Sept. 1990.
- [3] D. J. Miller and R. C. Lennox, "An Object-Oriented Environment for Robot System Architectures," IEEE International Conference on Robotics and Automation, 1990.
- [4] D. W. Forslund, C. Wingate, P. Ford, J. S. Junkins, J. Jackson, S. C. Pope, "Experiences in Writing a Distributed Particle Simulation Code in C++," USENIX C++ Conference Proceedings, San Francisco, CA, April 9-11, 1990.
- [5] R. J. Collins and D. R. Jefferson, "Selection in Massively Parallel Genetic Algorithms," Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann, 1991.
- [6] A. Baden, C. Day, R. Grossman, D. Lifka, E. Lusk, E. May, And L. Price, "A Data Model for Computations in High Energy Physics (preliminary report)," Laboratory for Advanced Computing Technical Report Number LAC91-R8, Univ. of Illinois at Chicago, December, 1991.
- [7] C. M. Chase, A. L. Cheung, A. P. Reeves and M. R. Smith, "Paragon: A Parallel Programming Environment for Scientific Applications Using Communication Structures," 1991 International Conference on Parallel Processing.
- [8] T. Keffer, "Object-Oriented Numerics, Part 1: Vectors, Matrices and All That Stuff," The C++ Journal, 1(4), 1991, pp. 3-9.
- [9] I. G. Angus, "Parallelism, Object Oriented Programming Methods, Portable Software and C++," Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp. 506-513.
- [10] D. W. Forslund, C. Wingate, P. Ford, J. Stephen Junkins, and S. C. Pope, "A Distributed Particle Simulation Code in C++," Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp. 514-518.
- [11] A. C. Robinson, A. L. Ames, H. E. Fang, D. Pavlakos, C. T. Vaughan, and P. Campbell, "Massively Parallel Computing, C++ and Hydrocode Algorithms," Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp. 519-526.
- [12] J. S. Peery and K. G. Budge, "Experiences in Using C++ to Develop a Next Generation Strong Shock Wave Physics Code," Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp. 527-534.
- [13] T. J. Ross, J. P. Morrow, L. R. Wagner and G. F. Luger, "Two Paradigms for OOP Models for Scientific Applications," Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp. 535-542.
- [14] T. J. Ross, L. R. Wagner and G. F. Luger, "Object-Oriented programming for Scientific Codes: Thoughts and Concepts," and "Object-Oriented Programming for Scientific Codes: Examples in C++," Univ. New Mexico Technical Report No. CS92-2, to appear in ASCE Journal of Computing in Civil Engineering.
- [15] J. M. Coggins, "C++ in Numerical and Scientific Computing," C++ Report, 4(3), 1992, pp. 65-68.
- [16] M. B. Stephenson, S. A. Canann and T. D. Blacker, "Plastering: A New Approach to Automated, 3D Hexahedral Mesh Generation", Sandia National Laboratories Report, SAND89-2192, February 1992.
- [17] T. Keffer, "Object-Oriented Numerics, Part 2: Virtual Algorithms," The C++ Journal, 2(2), 1992, pp. 3-8.
- [18] I. G. Angus, "An Object Oriented Approach to Boundary Conditions in Finite Difference Fluid Dynamics Codes," Scalable High Performance Computing Conference, 1992.
- [19] I. G. Angus, "Image Algebra: An Object Oriented Approach to Transparently Concurrent Image Processing," Scalable High Performance Computing Conference, 1992.

- [20] D. Quinlan, "Workshop on C++ for Scientific Computing", Abstracts in the proceedings of the SIAM Copper Mountain Conference on Iterative Methods, Copper Mountain, CO, April 9-14, 1992.
- [21] W. S. Brainerd, C. H. Goldberg and J. C. Adams, *Programmer's Guide to Fortran 90*, McGraw-Hill, 1990.
- [22] L.M. Taylor and D.P. Flanagan, "PRONTO-3D: A Three-Dimensional Transient Solid Dynamics Program," Sandia National Laboratories Report, SAND87-1912, 1989.
- [23] K.G. Budge, "PHYSLIB: A C++ Tensor Class Library," Sandia National Laboratories Report, SAND91-1752, 1991.
- [24] G. Walker, "Why the Choice Must Be C++," *The C++ Journal* 2(1), 1992, p. 52-65.
- [25] R. B. Davies, "Notes for the Library Working Group of WG21/X3J16," Presented at C++ Standards Committee Meeting, March 1991.
- [26] N. M. Wilkinson, "C++ Return Value Optimization," *The C++ Journal*, 2 (1), 1992, p. 47-51.

# **O-R Gateway: A System for Connecting C++ Application Programs and Relational Databases**

Abdallah Alashqur & Craig Thompson  
Central Research Labs  
Texas Instruments  
{alashqur, thompson}@csc.ti.com

## **Abstract**

Many application developers are adopting the use of object-oriented (OO) programming languages and design techniques to develop software applications because of the advantages these languages offer over conventional programming languages. The OO representation is also considered more powerful than the relational data model used to define relational databases. However, many enterprises have their data stored in databases managed under relational database management systems, and OO application programs often need to access this data to facilitate further processing. The two main problems facing OO application developers when accessing relational data directly are (1) they have to learn and handle two heterogeneous representation schemes in their programs, and (2) they do not get the full benefit of the object-oriented representation. To alleviate these problems, we need an approach and a system to bridge the gap between relational databases and object-oriented application programs by translating the relational definitions of data to equivalent OO definitions and enabling application developers to query relational data through these OO definitions. In this work, we describe an approach for achieving this and introduce the design of O-R Gateway, a system that enables C++ application programs to handle relational data as if they were C++ objects.

## **1. Introduction**

Many new software applications have been, and are being, developed using object-oriented (OO) programming languages and techniques. Applications in areas such as office information systems, CAD/CAM, CASE, and geographic information systems have requirements that can not be easily handled by traditional programming languages and design techniques. The type systems of OO programming languages (OOPL) encompass constructs that can be used to define complex objects, inheritance hierarchies, and behavioral properties of objects. An object state can be encapsulated, which enables updating the implementation of object classes without breaking application programs. These characteristics of OOPLs make them more desirable for handling many kinds of new applications than conventional programming languages.

Applications written in the OO programming language C++ need to interact with relational databases (RDBs) for several reasons. Some of these reasons are:

- 1. Existence of legacy data.** Many enterprises have their data stored in RDBs. This data is a necessary input to many decision making processes. Application programs written in C++ need to access this data in order to facilitate further processing.
- 2. Persistence.** Application developers often need to make some of the objects created in application programs persist between program invocations. The unavailability of wide-spread robust, scalable, and industrial strength OO DBMSs make relational DBMSs a viable candidate for maintaining persistent data generated by C++ application programs. Relational DBMSs are favored

over file systems, another candidate for storing persistent data, because they offer many useful functions such as concurrency control, recovery, physical data independence, and associative query capabilities.

**3. Data reverse engineering.** Once standard commercial OO DBMSs become available, many enterprises may want to migrate their legacy relational data to OO databases managed by these OO DBMSs. C++ application programs can be written to facilitate this process. These programs will need to access existing RDBs, retrieve data, construct objects by reformatting and assembling retrieved data, and then store these objects in an OO DBMS. This will automate the process of reverse engineering the data.

The current practice in accessing RDBs from C++ application programs is to wrap SQL queries in C functions and link these functions with the rest of the C++ code. The problem with this approach is that the application developer needs to learn and use two type systems, namely, the relational type system (relational data model) and the C++ type system, and he has to take care of translating the retrieved relational data to C++ objects. To alleviate this problem, a gateway is needed between the RDB and C++ environments that abstracts the details of the RDB away from the application developer. This gateway should enable C++ application developers to see an OO representation (objects, classes, inheritance relationships, etc.) of legacy data stored in relational databases, by translating a relational schema to equivalent C++ class definitions based on some mapping rules (see Section 3). These class definitions can be edited by a C++ programmer to add behavioral semantics (member functions) to them. The resulting class definitions can then be linked with C++ application programs that need to access the RDB.

An application developer who uses such a gateway does not have to be aware of the existence of the RDB. All he needs to know is that the instances of the classes he linked with his program are stored in some persistent store and they can be transferred from the store to the application program environment by means of set-oriented object queries. In other words, the queries used by the application developer are not directed against the relations in the RDB but against the C++ classes. The advantage of this is that the application developer has to know only one type system (i.e., C++), and the high-level object queries can take advantage of the inheritance property and the complex object definition facility of C++ (see Section 4). The gateway preprocesses the application program by translating the object queries to equivalent SQL queries against the RDB. The gateway then interacts with the RDB to retrieve relational tuples and translates them to C++ objects.

In this paper we describe the design and implementation of a gateway called the O-R Gateway that provides C++ application developers with a seamless accessibility to RDBs. Object queries that are supported in application programs that make use of O-R Gateway are easy to learn. They are syntactically upward compatible with SQL and include path expressions consisting of class object selectors (".") and class pointer selectors ("->") just as in C++ [LIP91]. O-R Gateway is an on-going project; in our implementation we use Oracle as the underlying DBMS, however, the ideas and techniques introduced in this paper are applicable to other DBMSs.

Several commercial and research prototype systems have employed certain methodologies and approaches for mapping from a high-level representation of data (semantic or Entity-Relationship models) down to a relational representation [LYN87, BLA88, TI90, PRE90]. O-R Gateway is



different from these systems in that its goal is to map from relational representation to high-level representation (based on the C++ type system) rather than the opposite. In [WIE80] and [NAV87], rules are described for mapping from a relational representation of data to the Structural data model and to the Entity-Relationship data model, respectively. In O-R Gateway, we make use of the mapping rules described in [WEI80] and further extend them to cover a wider variety of cases. PENGUIN [BAR90, BAR91] is a system that allows for defining objects on top of relational databases. These objects are similar to relational database views but with object attributes rearranged to remove redundancy and reflect the nesting of constituent object within more complex objects. Some major differences exist between O-R Gateway and PENGUIN, as described in Section 3.

This document is organized as follows. The overall architecture of O-R Gateway is presented in Section 2. In Section 3 we show how a relational schema can be mapped into a C++ schema. Query translation and object generation in O-R Gateway are described in Section 4. In Section 5 we give some conclusions and summarize planned future extensions to the O-R Gateway system.

## 2. O-R Gateway Architecture

In this section, we describe the overall architecture of the O-R Gateway system and briefly explain the functionality of its main modules.

Figure 1 illustrates how the Oracle C precompiler (which can be invoked using the command `pcc`) works. It accepts source files containing SQL statements embedded in C programs and translates the SQL statements into Oracle calls recognizable by the C compiler. To enable embedding SQL in C++ (SQL/C++), we have developed a processor called `pcc+` that edits the output of the Oracle precompiler to make it acceptable to the C++ translator. Figure 2 illustrates the data flow in the system when using SQL/C++ and `pcc+`. `Pcc+` was developed using the AWK programming language. One of the main tasks performed by `pcc+` is to use the linkage directive, `extern "C"`, with the Oracle C function calls generated by the Oracle precompiler. The AWK program that implements `pcc+` is Oracle-dependent, and therefore needs to be modified if another DBMS is used. However, this AWK program is very short (consists of three AWK rules) and can be easily replaced or modified. Thus, O-R Gateway is highly portable.

Figure 3 illustrates where the O-R Gateway fits in the overall architecture. As described in the figure, a programmer can either write SQL/C++ or OQL/C++ application programs, where OQL is an object query language designed to operate on the constructs of the C++ type system. O-R Gateway takes care of the schema, query, and object translations between the C++ program and the underlying relational DBMS, therefore, bridging the impedance mismatch between their type systems. SQL/C++ programs can be written by those programmers who are familiar with the relational type system (data model) as well as the C++ type system and do not want to take advantage of O-R Gateway. An SQL/C++ programmer has to explicitly take care of translating data representations between the two type systems. On the contrary, an OQL/C++ programmer only sees the C++ type system and accesses persistent data via high-level object queries issued against C++ classes. The relational DBMS and data model are transparent to this programmer.

Figure 4 illustrates the detailed architecture of the O-R Gateway system. The path that starts at the OO Schema Generation Module (SGM) and ends at the Library Archives translates relational

schemas to equivalent C++ schemas with added behavioral semantics. The SGM extracts information about the schema of a relational database from the relational dictionary. Based on some mapping rules (see Section 3) and user or database administrator (DBA) input, SGM generates some header files that contain the C++ class definitions to represent the underlying relational data. After the SGM, the class definitions only contain a structural component (i.e., data members and type-subtype relationships) but do not include a behavioral component (i.e., member functions). A user (programmer) can edit these header files to add member functions declarations and can add new files containing member functions definitions. The resulting files can then be compiled by the C++ compiler and stored in Library Archives.

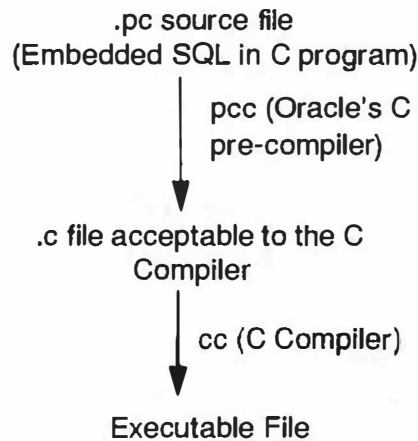


Figure 1 Processing Embedded SQL in C by the Oracle Precompiler

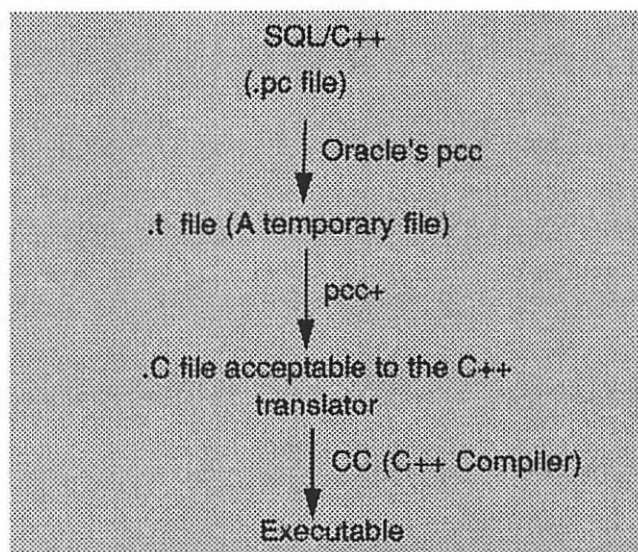


Figure 2. Extending the Capability of Oracle's pre-Compiler to handle C++

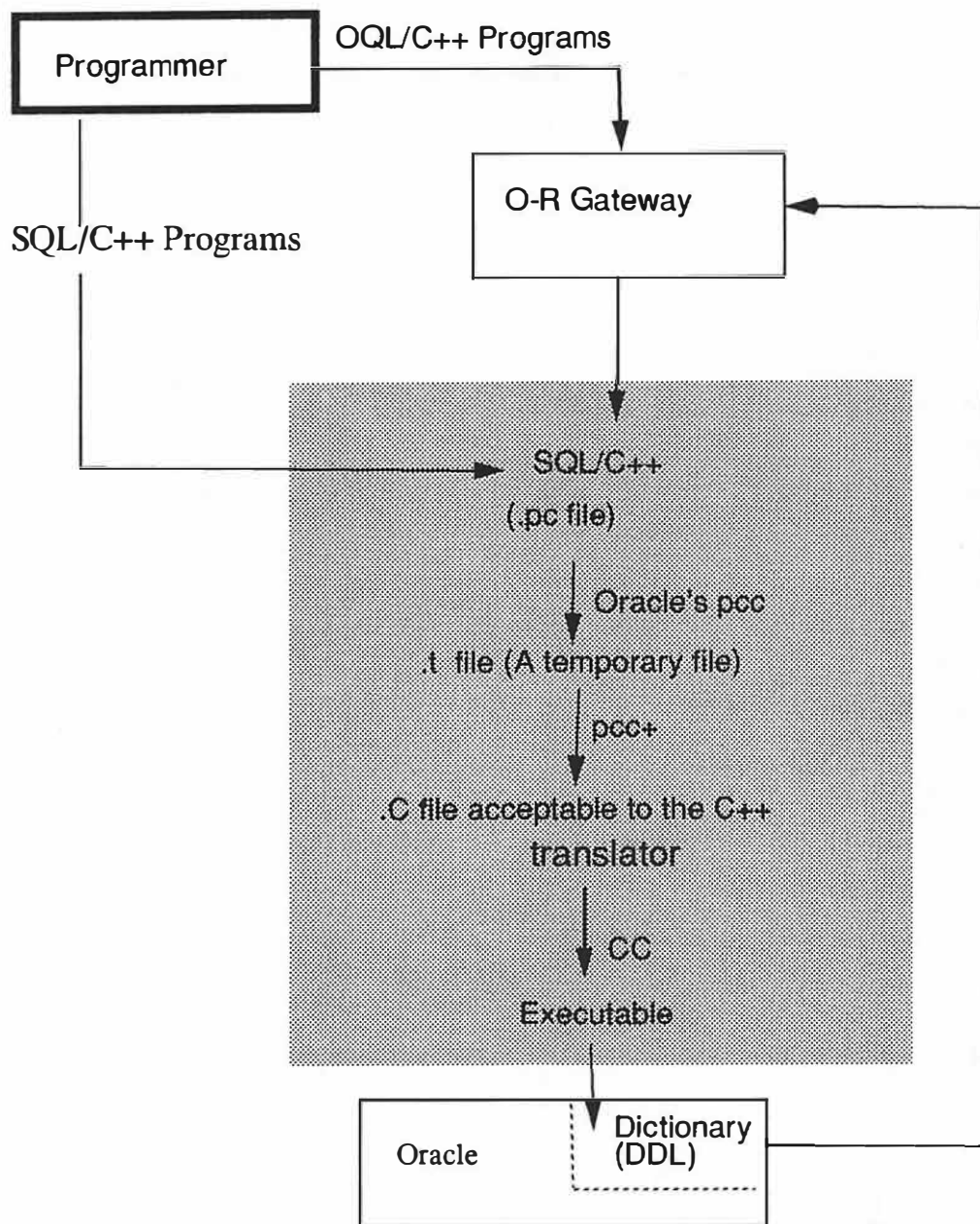


Figure 3. O-R Gateway Enables Object Queries in C++ Programs

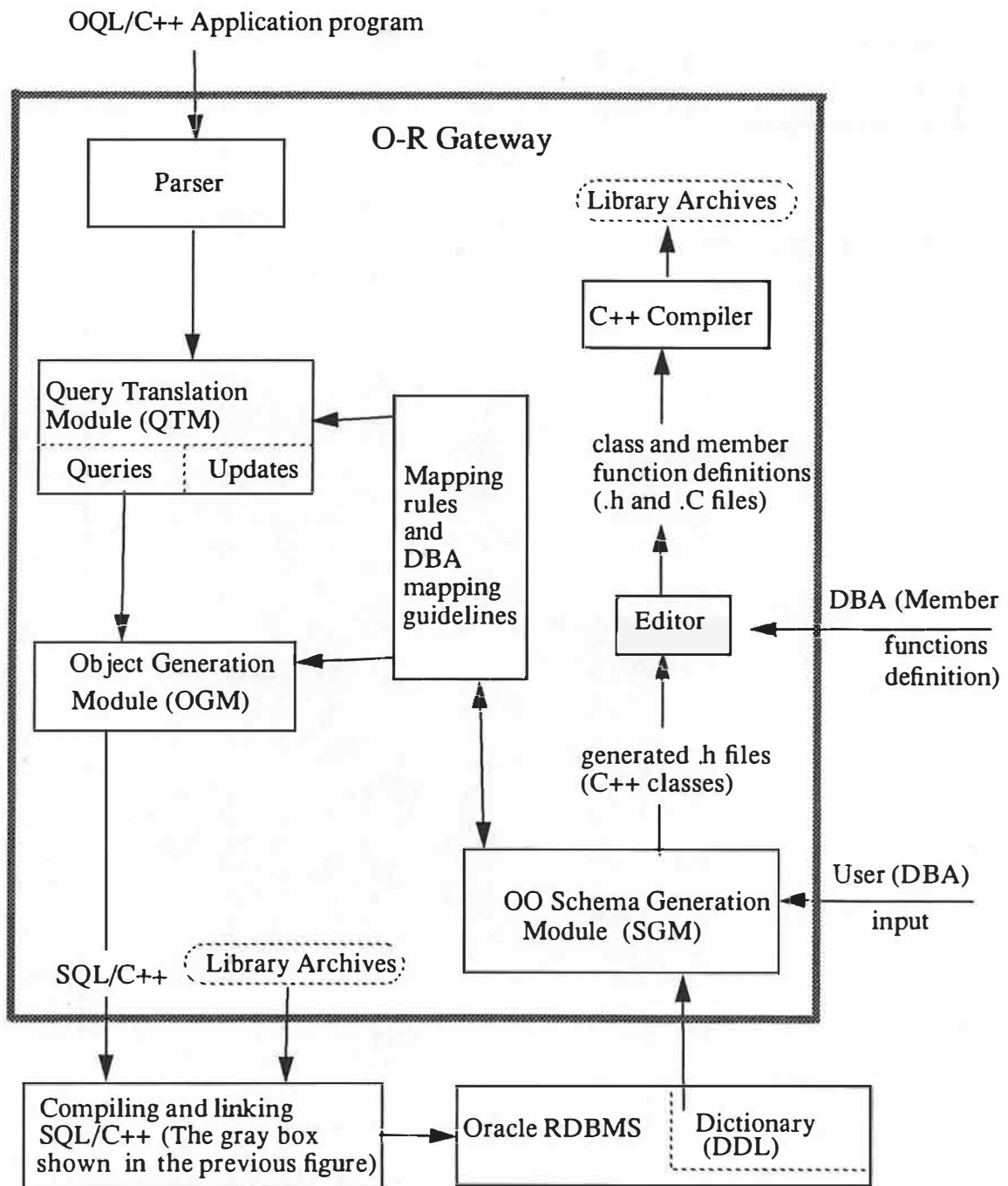


Figure 4. O-R Gateway Modules

The path that starts at the Parser and ends at SQL/C++ in Figure 4, takes care of translating object queries to SQL, and of generating C++ objects out of retrieved data. Header files generated by the SGM and edited by the user, that correspond to a particular database, need to be *included* in OQL/C++ application programs that use the database.

The Parser syntactically validates the object query statements and generates internal data structures for representing these statements. The Query Translation Module (QTM) translates parsed object query statements to equivalent SQL statements that are targeted to the underlying relational database and adds the necessary CONNECT statements to connect to the relational DBMS (see Section 4).

The Objects Generation Module (OGM) generates code that constructs C++ objects by assembling relevant data retrieved from the relational database. The mapping rules and DBA guidelines used by the SGM are also used by QTM and OGM. The library archives that include the compiled C++ type definitions and member functions are linked with the generated SQL/C++ program. The library archives and header files for a database need to be generated only once, but can be used by several OQL/C++ application programs that need to access the database.

### 3. Translating Relational Schemas to C++ Type Definitions

In this section, we demonstrate how a relational schema can be mapped into a schema represented in the C++ type system. We make use of the rules employed in PENGUIN [BAR91] for mapping a relational schema to a schema represented in the structural data model [WIE80, BAR91]. However, there are major differences between the O-R Gateway system and PENGUIN. The following are some of these differences. (1) We modify the definition of some connection types (see Section 3.1 below for a definition of the term *connection type*) and define other new connection types in order to be able to address important cases not covered in PENGUIN. (2) We show how an OO schema can be represented using the type system of a specific OOPL (C++). (3) O-R Gateway allows a user or a DBA to add a behavioral component (operations or methods) to the resulting C++ schema. (4) PENGUIN provides the application programmer (or end user) with a set of predefined view objects that the user can query. O-R Gateway, on the other hand, presents the application programmer with the entire database schema represented in the C++ type system and the application programmer is free to pose queries against any class.

#### 3.1 Connection Types in the Structural Data Model

In the structural data model, a connection exists between two relations  $R_1$  and  $R_2$  based on two subsets of their attributes  $A_1$  of  $R_1$  and  $A_2$  of  $R_2$ , if data types of  $A_1$  attributes are identical to those of  $A_2$  and the cardinality of  $A_1$  is equal to the cardinality of  $A_2$  (i.e.,  $A_1$  and  $A_2$  are union-compatible).  $A_1$  and  $A_2$  are referred to as the *connection attributes*. Two tuples  $t_1$  and  $t_2$  of  $R_1$  and  $R_2$ , respectively, are connected if the values of  $A_1$  attributes in  $t_1$  is equal to the values of  $A_2$  attributes in  $t_2$ , i.e.,  $t_1[A_1] = t_2[A_2]$ , where  $t[A]$  denotes the sub-tuple of  $t$  that contains the  $A$  attribute values.

Three types of connections were identified in [WIE80, BAR91]. In the following, we define each of the three types of connections in terms of primary and foreign key constraints on relations. We use  $PK(R)$  to denote the set of attributes that constitute a primary key of relation  $R$  and  $FK(R)$  to

denote the set of attributes that constitute a foreign key of R. We also use A1 and A2 to denote the connection attributes of relations R1 and R2, respectively. (Note: the set of foreign key attribute values in a relation must be subset of the set of values of the primary key attribute of another relation [DAT81, ELM89].)

Each of the three connection types must satisfy the following rule:

$A1 = PK(R1)$  and  $A2 = FK(R2)$ .

In addition to this common rule, a specific rule must be satisfied by each connection type as defined below:

1. An *ownership connection* from R1 to R2 exists if  $A2 \subset PK(R2)$  ( $A2$  is a subset of the set of primary key attributes of R2).
2. A *subset connection* from R1 to R2 exists if  $A2 = PK(R2)$ .
3. A *reference connection* from R2 to R1 exists if none of the rules for the other connection types is satisfied (i.e., only the common rule stated above is satisfied).

To demonstrate, we apply the above rules to the following relational schema and constraints:

DEPARTMENT (dname, floor, budget)  
EMPLOYEE (ename, dept, project)  
HOURLY\_EMP (ename, wage, skills)  
EMP\_CAR (ename, carno, decal\_date)

Primary keys are underlined. Foreign key constraints on these relations are: dept in EMPLOYEE is a foreign key matching dname in DEPARTMENT and ename is a foreign key in both HOURLY\_EMP and EMP\_CAR matching ename in EMPLOYEE. carno in EMP\_CAR identifies an employee's cars (i.e., first, second, etc.) and is not unique across all tuples of the relation.

Connections that exist between these relations based on the rules given above are:

1. Ownership connection. Since ename is the primary key of EMPLOYEE and is part (subset) of the primary key of EMP\_CAR, and given the above foreign key constraints, there is an ownership connection from EMPLOYEE to EMP\_CAR. EMP\_CAR tuples are owned by EMPLOYEE tuples in the sense that an EMP\_CAR tuple cannot exist in the database without being related to an EMPLOYEE tuple. The multiplicity (cardinality) from an owner relation to an owned relation is 1:n; an EMPLOYEE tuple can own several EMP\_CAR tuples, while an EMP\_CAR tuple must be owned by only one EMPLOYEE tuple.
2. Subset connection. There is a subset connection from EMPLOYEE to HOURLY\_EMP based on the connection attribute ename, which is the primary key of both EMPLOYEE and HOURLY\_EMP and a foreign key in HOURLY\_EMP. This implies that the set of entities that HOURLY\_EMP tuples describe is a subset of the set of entities described by the tuples of EMPLOYEE.
3. Reference connection. There is a reference connection from EMPLOYEE to DEPARTMENT based on connection attributes dept and dname, since none of the rules for ownership and subset connections is satisfied. An EMPLOYEE tuple is said to reference a DEPARTMENT tuple. The

multiplicity of a reference connection is n:1; an EMPLOYEE tuple can reference only one DEPARTMENT tuple, while a DEPARTMENT tuple can be referenced by several EMPLOYEE tuples.

### 3.2 Extensions

In this subsection, we extend the definition of some of the connection types described above for the structural data model and define some new connection types. Our goal is to cover cases that may exist in a relational schema that are not covered by the above three connection types. In what follows, we first extend the definition of the ownership connection and then define some new connection types.

1'. Ownership connection (generalized). In some cases, a foreign key A2 of a relation R2 (matching A1 of R1) is not part of the primary key of that relation, yet, there is an existence dependency of tuples of R2 on tuples of R1 expressed as a NON NULL constraint on A2. This case becomes clear if we replace carno of EMP\_CAR with registrationNo whose values are unique across all tuples. The definition of EMP\_CAR becomes:

EMP\_CAR (registrationNo, ename, decal\_date)

where registrationNo is the primary key and ename is a foreign key which is constrained to be NON NULL. An EMP\_CAR tuple can not exist in the database without being related to an EMPLOYEE tuple. This is because an EMP\_CAR tuple has to contain an ename value (ename is NON NULL) and this value has to exist in EMPLOYEE (due to the foreign key constraint). Therefore, there is an ownership connection from EMPLOYEE to EMP\_CAR. To handle this case, we modify the definition of an ownership connection type to the following:

An ownership connection from R1 to R2 exists if  $A2 \subset PK(R2)$  OR  $NON\_NULL(A2)$ .

Below, we introduce the definition of two new connection types.

4. Set equality connection. A *set-equality connection* exists between R1 and R2 if  $A2 = PK(R2)$  AND  $A1 = FK(R1)$  AND  $A1=PK(R1)$  AND  $A2=FK(R2)$ .

The last two predicates in this list of predicates is the common rule that the above three connection types must satisfy. This connection type implies that the set of entities described in R1 must be equal to the set of entities described in R2. A practical example of this case can be illustrated if we define the relation:

EMP\_INSU (ename, policy, date)

which adds life insurance information about employees to the above database. To satisfy a requirement that every employee in the company must have life insurance, then ename in EMP\_INSU should be modeled as a foreign key attribute matching ename in EMPLOYEE, and ename in EMPLOYEE should also be modeled as a foreign key matching ename in EMP\_INSU.

5. Set intersection connection. A *set-intersection connection* exists between R1 and R2 if  $A1=PK(R1)$  and  $A2=PK(R2)$ . As a special case, the common rule that is applicable to the above four connection types, does not apply to this connection type.



A set-intersection connection between two relations implies that the two sets of entities described in the two relations do not have to be equal and their intersection may or may not be null.

### 3.3 Mapping to C++ Type Definitions

In this subsection, we describe a procedure for generating a C++ schema out of a relational schema. We use the following relational schema to illustrate this procedure, assuming that the foreign key and NON-NULL constraints described above are applicable.

```
DEPARTMENT (dname, floor, budget)
EMPLOYEE (ename, dept, project)
HOURLY_EMP (ename, wage, skills)
EMP_CAR (registrationNo, ename, decal_date)
```

The generated C++ schema is shown in Figure 5. An equivalent schema represented graphically is shown in Figure 6, where small circles denote classes, and thick and thin links denote type-subtype and data member (attribute) relationships, respectively. The following is a procedure that consists of seven steps to map relations and connection types to C++ constructs.

1. Create a class corresponding to each relation. For simplicity, we use relation names preceded by the string "C\_" to denote class names. Accordingly, the classes corresponding to the above relational schema are C\_DEPARTMENT, C\_EMPLOYEE, C\_HOURLY\_EMP, and C\_EMP\_CAR (Figure 5).

2. Each non-foreign key attribute of a relation becomes a data member of the corresponding class (e.g., dname, floor, and budget attributes of DEPARTMENT become data members of C\_DEPARTMENT). The data types of the data members defined in this step are selected from the built-in C++ types (int, char, float, etc.). If the type of a data member is one of the C++ built-in types, it must be as close as possible to the datatype of the corresponding relational attribute. Since, normally, the set of types in a relational DBMS does not exactly match the set of C++ built-in types, type conversion needs to be performed when objects are retrieved. (In case of Oracle, this type conversion is performed by the Oracle C Pre-compiler.)

3. We use the terms *referencing* and *referenced class* to denote classes corresponding to a referencing and referenced relations, respectively. There is a reference connection from relation EMPLOYEE to relation DEPARTMENT, therefore, C\_EMPLOYEE is a referencing class and C\_DEPARTMENT is a referenced class. In this step of the procedure, a data member is created in every referencing class. The type of this data member is a **pointer** to the referenced class. Hence, in Figure 5, the data member dept of C\_EMPLOYEE is defined as a pointer to C\_DEPARTMENT.

4. We use the terms *owner* and *owned class* to refer to classes corresponding to an owner and owned relations, respectively. Therefore, C\_EMPLOYEE is an owner class and C\_EMP\_CAR is an owned class. In this step, a data member is created for every owner class. The type of this data member is "SET of" the owned class. Hence, the data member cCars of C\_EMPLOYEE is defined as SET of C\_EMP\_CAR.



5. A subset connection between two relations maps to a type subtype relationship between the classes corresponding to the two relations. Therefore, C\_HOURLY-EMP is defined in Figure 5 as a subtype of C\_EMPLOYEE.

6. For any two classes corresponding to two relations connected by a set-equality connection, a new class is created to act as a super class of these two classes. The data members common to the two classes are moved up to the super class.

7. Any two classes corresponding to two relations connected by a set-intersection connection are handled in the same way as step 6 above.

Another possible mapping for step 6 above is to merge the two classes into one class whose set of attributes is the union of the two sets of attributes of the two classes. For example, C\_EMPLOYEE and C\_EMP\_INSU classes can be merged to form one class whose data members are `ename`, `dept`, `project`, `policy`, and `date`. However, we keep the scheme described in step 6 as the default option and provide the DBA with the capability of interacting with the SGM module (Figure 4) to specify other mapping options. The advantage of this is that step 7 and the default option in 6 can be treated uniformly, which simplifies the initial prototype implementation. (Note that merging the two classes in step 7 will result in losing some semantic information about objects.)

In addition, a database designer input is necessary during the mapping process for the following reasons. (1) To choose the names of classes and data members in the C++ schema. By default, data member names are the same as the attribute names, and class names are the same as the relations names preceded by the string "C\_". A user (or a DBA) may prefer to choose different names. In this case, a name manager (part of O-R Gateway) stores the mappings between the C++ schema names and the underlying relational schema names. Query processing makes use of these name mappings. (2) To add the definition of necessary public member functions to the generated C++ schema. These functions define the behavior of the C++ objects. (3) To provide any information that is missing from the underlying relational DBMS (not all DBMSs support all functionalities) such as foreign-key relationships. The mapping rules and the DBA guidelines and choices will be recorded in the system since they will also guide the operation of the QTM and OGM modules (see Figure 4).

In the above procedure, all generated data members are declared as *private*. For each generated class, a *friend* class called `db_class` is declared (Figure 5). Since the application developer needs to access the private data members to query objects (i.e., transfer objects from the persistent store to C++ environment), he needs to place such object queries in public member functions of the class `db_class`. Therefore, data encapsulation is preserved.

In the current implementation status, schema mapping is performed manually according to the above mapping rules. Schema mapping modules (Figure 4) will be included in a forthcoming version of O-R Gateway. However, a basic query translation capability (see Section 4) has been implemented and tested.

```

/* define a parametrized set class */
template <class type>
class SET {
/* .. data members and member functions definitions ..*/
}

class C_DEPARTMENT {
friend class db_class;
private:
char *dname;
int floor;
float budget;
};

class C_EMP_CAR {
friend class db_class;
private:
float registrationNo;
char* decal_date;
};

class C_EMPLOYEE {
friend class db_class;
private:
char *ename;
C_DEPARTMENT *dept;
SET<C_EMP_CAR> cars;
char *project;
};

class C_HOURLY_EMP: public C_EMPLOYEE {
friend class db_class;
private:
float wage;
char *skills;
};

```

Figure 5: a C++ schema

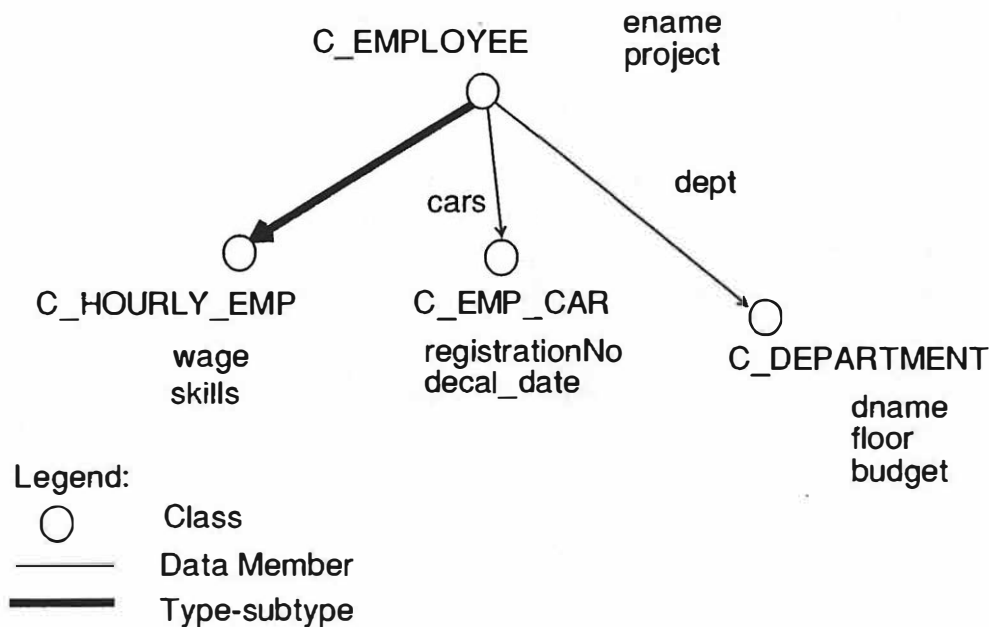


Figure 6. An OO Schema as a Graph

#### 4. Translating Object Queries to SQL Queries

One of the key features of object queries as supported by many existing object query languages [ALA90, BLA90] is the use of path expressions. If Class1 has a data member whose type is Class2 and Class2 has a data member whose type is Class3 in some schema, then "Class1.Class2.Class3" is a path expression that starts at Class1 and ends at Class3. Path expressions enable logical navigation at the schema level and can be used in specifying predicates or identifying the list of attributes to be retrieved. For example, "Class1.Class2.Class3 == value" is an associative predicate that identifies all the Class1 objects whose related Class3 objects are equal to the given value. Path expressions are used for querying complex objects by specifying predicates on data members that are deeply nested within the structure of these objects.

In O-R Gateway, we support the use of object queries that have the following structure in C++ programs:

```

SELECT <class-name> or
    <path expressions rooted at a single class>
FROM <range variable declarations>
WHERE <predicates that may involve path expressions>.
  
```

Where the FROM clause is optional. This syntax is similar to that of OQL[C++] developed at Texas Instruments [BLA90] for the Zeitgeist object-oriented database system [FOR88]. The effect of a query is to transfer a set of objects from the persistent object store to the C++ environment. The

fact that the persistent object store is a relational database is transparent to the application developer.

#### 4.1 Query Translation Module (QTM)

To translate an object query to an SQL query, we need to translate path expressions and expressions involving inheritance relationships to equivalent SQL joins. These joins are over matching key and foreign key attribute values of the relations corresponding to the classes referenced in the object query. A FROM clause in the SQL query will list the referenced relations. The Query Translation Module QTM performs such translations.

For illustration, the following are four example object queries and their equivalent SQL queries. The first two of these queries contain path expressions in the WHERE and SELECT clauses, respectively. The third query demonstrates how a FROM clause in an object query can be useful, and the fourth query demonstrates how inheritance is handled. (Example queries in this section are expressed against the relational schema and its equivalent C++ schema described in Section 3 of this paper.)

##### Query 1:

```
SELECT C_EMPLOYEE.ename, C_EMPLOYEE.project
WHERE C_EMPLOYEE.dept->floor = 2;
```

Conceptually, the WHERE clause of this query returns all the C\_EMPLOYEE objects for employees that work in departments located in the second floor. The final "answer set" of the query contains these objects and values for their ename and project data members. Values of other data members are not included in the answer set since they are not SELECTed by the query. QTM translates this query to the following SQL query:

```
SELECT EMPLOYEE.ename, EMPLOYEE.project
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.dept = DEPARTMENT.dname AND
      DEPARTMENT.floor = 2;
```

The path expression "C\_EMPLOYEE.dept" in the WHERE clause of the object query is translated to the SQL join predicate "EMPLOYEE.dept = DEPARTMENT.dname". Syntactic simplification can be made in both object and SQL versions of a query by removing redundant information. For example, the attributes need not be qualified by their relation names in the SELECT clause of the above SQL query, therefore, reducing it to "SELECT ename, project". This is because each of ename and project is an attribute of exactly one of the relations referenced in the FROM clause. However, in this paper, we use full syntax for the sake of clarity.

##### Query 2:

This query demonstrates the use of path expressions in the SELECT clause.

```
SELECT C_EMPLOYEE.ename, C_EMPLOYEE.dept->budget
WHERE C_EMPLOYEE.project = 'OODB';
```

Translates to the following SQL query where a join predicate is added to the WHERE clause to account for the path expression:

```
SELECT EMPLOYEE.ename, DEPARTMENT.budget
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.dept = DEPARTMENT.dname AND
      EMPLOYEE.project = 'OODB'
```

### Query 3:

This object query demonstrates one of the ways in which the optional FROM clause can be used (note that the FROM clause is not optional in SQL).

```
SELECT E1.ename, E2.ename
FROM C_EMPLOYEE E1 E2
WHERE E1.project = E2.project
```

This query selects pairs of employee names for employees co-working on the same project. E1 and E2 are two range variables whose type is C\_EMPLOYEE. This query is translated to the following SQL query:

```
SELECT E1.ename, E2.ename
FROM EMPLOYEE E1, EMPLOYEE E2
WHERE E1.project = E2.project
```

### Query 4:

Since a subclass inherits the members of its superclass, an object query can reference the inherited data members directly. The following is an example query that involves inheritance.

```
SELECT C_HOURLY_EMP.project, C_HOURLY_EMP.wage
WHERE C_HOURLY_EMP.wage > 10k
```

C\_HOURLY\_EMP inherits project from C\_EMPLOYEE, therefore project is referenced in the SELECT clause as an ordinary data member of C\_HOURLY\_EMP. The following SQL query accounts for this by performing an equijoin between the relations EMPLOYEE and HOURLY\_EMP over ename values.

```
SELECT EMPLOYEE.project, HOURLY_EMP.wage
WHERE EMPLOYEE.ename = HOURLY_EMP.ename AND
      HOURLY_EMP.wage > 10k
```

The Query Translation Module, QTM, (Figure 4) uses the schema translation rules and DBA guidelines that are used by the Schema Generation Module (Section 3) in the process of translating an object query to SQL. QTM will, based on these transformation rules, identify project as an attribute of EMPLOYEE and not of HOURLY\_EMP relation even though it is referenced as a data member of the C\_HOURLY\_EMP class in the object query. QTM will also, based on these transformation rules, identify ename as the attribute that links the two relations EMPLOYEE and

DEPARTMENT, and therefore it generates the predicate "EMPLOYEE.ename = HOURLY\_EMP.ename" as part of the WHERE clause of the SQL query.

Object queries need to access private data members of classes in order to retrieve objects from the persistent store. In order to preserve data encapsulation, application developers need to place object queries in public member functions of the class `db_class`, which is declared as a friend class to all the classes representing persistent objects (see Section 3).

## 4.2 Object Generation Module (OGM)

If an object query is included in an OQL/C++ program that is processed by O-R Gateway, the translated SQL query will appear in the generated SQL/C++ program preceded by the necessary host variable declarations and Oracle connect statements, and will be followed by code that constructs C++ objects out of the retrieved relational data. This is performed by the OGM, which generates the code that assembles data retrieved from the relational database to construct complex C++ objects.

An OQL/C++ application programmer needs to declare an aggregate object type such as an array or a set (or a pointer to an aggregate object) to hold the result of an object query. The data type of the elements of the aggregate object is either the class whose objects are to be retrieved or a pointer to it. Therefore, Query 1 above would actually be written in an OQL/C++ application program as follows.

```
/* EMPAR is declared as a pointer to an array of pointers to C_EMPLOYEE objects */
C_EMPLOYEE** EMPAR = new C_EMPLOYEE* [100];

EMPARE =
    SELECT C_EMPLOYEE.ename, C_EMPLOYEE.project
    WHERE C_EMPLOYEE.dept->floor = 2;
```

This query will assign pointers to the first 100 C\_EMPLOYEE objects retrieved to the cells of the array EMPAR. Once objects are retrieved they can be processed in the rest of the C++ program in a normal way.

Alternatively, using C++ Class Templates definition capability (C++ Version 3.0 [LIP91]), one can declare a set whose type is *pointer* to C\_EMPLOYEE objects as follows.

```
/* SET is declared somewhere else in the program as a Template Class */
SET<C_EMPLOYEE*> *EMPSET = new SET<C_EMPLOYEE*>;

EMPSET =
    SELECT C_EMPLOYEE.ename, C_EMPLOYEE.project
    WHERE C_EMPLOYEE.dept->floor = 2;
```

EMPSET can then be used to hold pointers to the employee objects returned by the query<sup>1</sup>.

---

1. In our current implementation of O-R Gateway we have used aggregate objects of type ARRAY and not SET since we are using an earlier version of the C++ compiler in which class templates are not supported.

The Object Generation Module (OGM) will add the necessary code that creates the C++ objects, assigns attribute values retrieved from the database to data members of objects, and adds the necessary CONNECT statements to connect to Oracle. The C++ code generated by the QTM and OGM to implement the above query and assign the result to the array EMPAR is shown below.

```
EXEC SQL BEGIN DECLARE SECTION;

    VARCHAR uid[20];
    VARCHAR pwd[20];
    VARCHAR ename_var[15];
    VARCHAR project_var[15];
    int floor;

EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca.h;
EXEC SQL INCLUDE oraca.h;

strcpy(uid.arr,"USERNAME");
uid.len = strlen(uid.arr);
strcpy(pwd.arr,"PASSWORD");
pwd.len = strlen(pwd.arr);

EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
EXEC SQL DECLARE emp_objects CURSOR FOR

SELECT EMPLOYEE.ename, EMPLOYEE.project
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.dept = DEPARTMENT.dname AND
      DEPARTMENT.floor = 2;

EXEC SQL OPEN emp_object;
EXEC SQL WHENEVER NOT FOUND GOTO end_of_fetch;

/*retrieve data for the first 100 employee objects */
for (int i = 0; i < 100; ++i) {

EXEC SQL FETCH emp_object INTO :ename_var, :project_var;
ename_var.arr[ename_var.len] = '/0';
project_var.arr[project_var.len] = '/0';

/*create the i'th employee objects */
EMPAR[i] = new C_EMPLOYEE;

/* assign attribute values to corresponding object data members */
EMPAR[i] -> ename = ename_var;
EMPAR[i] -> project = project_var;
} /* end of for */
end_of_fetch:
EXEC SQL CLOSE emp_object;
EXEC SQL COMMIT WORK RELEASE;
/* log off the Oracle database. */
```

## 5. Conclusion

In this paper, we described the design of the O-R Gateway system, which provides an object-oriented veneer on top of a relational database. O-R Gateway enables C++ programmers to view and access relational databases as if they were object repositories. The overall architecture of the system and the main functions of each of its modules were described. A process for mapping a relational schema to C++ type definitions was presented. The way object queries can be used in a C++ program to retrieve objects from the database was described.

Our future plan includes the addition of the following enhancements to O-R Gateway: (1) Enhance the existing capabilities of the Query Translation and Object Generation Modules, (2) Implement the Schema Generation Module, (3) Support update operations against objects, (4) Support simultaneous access to other relational database systems such as DB2 and Ingres from C++ application programs, (5) Support mapping a C++ schema to a relational schema, which is the inverse of the mapping described in Section 3, and (6) Develop a scalable, robust system.

## Bibliography

- ALA90 A. Alashqur, S. Su, and H. Lam, "A Rule-based Language for Deductive Object-oriented Databases," Proceedings of the 6th IEEE Intl. Conf. on Data Engineering, 1990.
- BAR90 T. Barsalou and G. Wiederhold, "Complex Objects for Relational Databases," Computer-Aided Design, October, 1990.
- BAR91 T. Barsalou, N. Siambela, A.M. Keller, and G. Wiederhold, "Updating Relational Databases through Object-Based Views," 1991 ACM SIGMOD Conference on Management of Data Proceedings.
- BLA88 M. Blaha, W. Premerlani, and J. Rumbaugh, "Relational Database Design Using and Object-Oriented Methodology," Communications of the ACM, April, 1988.
- BLA90 J. Blakeley, C. Thompson, and A. Alashqur, "Strawman Reference Model for Object Query Languages," Proceedings of the X3/SPARC/DBSSG OODB Task Group Workshop on Standardization of Object-Oriented Database Systems, Atlantic City, N.J., May 22, 1990. A revised version has also appeared in the Proceedings of the International Journal on Computer Standards and Interfaces, 1991.
- DAT86 C. Date, An Introduction to Database Systems, Vol. 1, Fourth Edition, Addison-Wesley, Menlo Park, CA, 1986.
- DAT81 C.J. Date, "Referential Integrity," Proceedings of the VLDB Conference, France, 1981.
- ELM89 R. Elmasri and S. Navathe, Fundamentals of Database Systems, The Benjamin/Cummings Publishing Company, Inc., 1989.
- FOR88 Steve Ford, et al., "Zeitgeist: Database Support for Object-oriented Programming," in the Proc. of the 2nd Intl. Workshop on Object-oriented Database Systems, 1988.
- LIP91 Stanley Lippman, C++ Primer, Addison-Sesley Publishing Company, 1991.



- LYN87 Peter Lyngbaek and Victor Vianu, "Mapping a Semantic Database Model to the Relational Model," 1987 ACM SIGMOD Conference on Management of Data Proceedings, Pages 132-142.
- NAV87 S.B. Navathe and A.M. Awong, "Abstracting Relational and Hierarchical Data With a Semantic Data Model," Proceedings of the 6th International Conference on Entity-Relationship Approach, New York, Sal March (ed.), 1987.
- PRE90 W.J. Premerlani, M.R. Blaha, J.E. Rumbaugh, and T.A. Varwig, "An Object-Oriented Relational Database," Communications of the ACM, November 1990.
- RUM87 J. Rumbaugh, "Relations as Semantic Constructs in an Object-Oriented Language," OOPSLA 87 Conference Proceedings.
- TI90 Texas Instruments Incorporated, A Guide to Information Engineering Facility Using the IEF, a book, Second Edition.
- WIE80 G. Wiederhold and R. Elmasri, "The Structural Model for Database Design," Proceedings of the International Conference on Entity-Relationship Approach to Systems Analysis and Design, 1980.



# Static Initializers: Reducing the Value-Added Tax on Programs

John F. Reiser  
*Mentor Graphics Corporation*  
Wilsonville, Oregon 97070 USA  
jreiser@mentorg.com

## ABSTRACT

The declaration of a C++ initialized static object in a .h header file to provide initialization via “allocation hook” acts much like a value-added tax: everyone pays, up and down the line. The software writer, the module integrator, and the maintenance engineer all pay. Even the end user of the software pays: invoking and terminating large systems can take dozens of seconds solely because of allocation hook static objects. This paper relates experience dealing with static initialization in several large software applications (thousands of classes, tens of thousands of functions, millions of instructions). Poor locality of instruction pages [large working set size] accounts for the runtime cost of static initialization. A facility for address tracing in real time, with on-line interactive graphical visualization, aids the exploration of how to reduce such costs.

## The need for initializers

Consider the C++ code fragment in Figure 1. The compiler can initialize the integer with a compile-time constant. The compiler can direct the static linker to initialize the pointer with a link-time constant (or, the linker will delegate the job to the instantiator of executable images within a process.) The compiler must generate code to arrange for the initialization of the class object. Generating code is no problem, but arranging for execution of the code is a real problem. The execution of such initialization code is implicit and out-of-line compared to statements within member functions.

```
int const x = 5; // compile-time constant
int *const px = &x; // link-time constant
class Foo {
private: int x;
public: Foo(int);
};
Foo const foo(5); // initialization requires execution
```

Figure 1. Initializations

For another case, consider the programmer who wishes to implement an application programming interface (API) using C++ classes. The programmer publishes the class declarations in file `api.h`, which then gets `#included` by all clients of the interface. If the implementation of the API requires runtime constants such as an open file or socket, then the programmer has at least three implementation choices. First, the programmer may require that the client call a particular function before touching any other part of the API. Such a requirement shifts the initialization burden onto the enclosing API that the client is implementing itself. Or, second, the programmer may guard the beginning of each function in the API with code `if (!init) init();` This does not work if the API contains data members, and can be expensive in time and space for small functions that are called frequently. Or, third, the programmer may declare in `api.h` an initialized static class instance [not lexically visible outside the current compilation unit] whose sole purpose

is to “piggyback” the initialization of the API onto the initialization provided by C++ for the static class instance. This method is called “allocation hook” initialization [ARM90 pp20-21 attributes this to Jerry Schwarz], since it relies on the allocation [and construction] of something else to achieve initialization of the API. In the AT&T cfront C++ language system, class `iostream_init` `#include <iostream.h>` uses initialization by allocation hook to link up `cin`, `cout`, and `cerr` with corresponding input/output support structures. Cfront uses this mechanism to achieve portability of implementation, since the C language (into which cfront translates its C++ input) has no architected facility for runtime initializations. Cfront puts the statements that are necessary to perform the initializations requested by the current compilation unit into a function whose name begins with `__sti__`, then arranges through mechanisms outside the language that all the `__sti__` functions are called at the beginning of execution. These functions are called *static initializer functions*, where ‘static’ refers to the fact that the space occupied by the initialized class instance is allocated before execution begins, and is not dynamic.

### Benefits and costs of static initializers

Static initializer functions enable cfront to accomplish initialization using portable C code (although the extra-language mechanisms may require adaptation to the platform’s native compilation system.) While providing portability, static initializer functions have costs. Every compilation unit (object file) whose source code directly or indirectly `#include <iostream.h>` receives several mementos of the tryst. Of course, there is the initialized static object itself (2 ints). Then there are two functions (the `__sti__` and `__std__` functions) to supervise the object, and a ball and chain (the `__link` structure: 3 pointers) for patch linking [one of the extra-language mechanisms]. In software products with few compilation units, these overheads usually can be ignored since they are either absolutely small, or they are a small-enough fraction of the total. In a larger system, the overheads are no longer absolutely small, and their relative impact can be larger still.

Mentor Graphics Corporation constructs software in C++ for several large electronic design automation applications. Each piece (shared library, dynamically-loaded module, mainline application) typically contains hundreds of files, hundreds or thousands of classes, and millions of instructions. Much of the source code is written in an object-oriented style, with classes that have many small member functions. Table 1 gives some summary statistics for five modules. Although the C++ language has been popular for several years, full-featured native C++ compilation systems are still not universal on the engineering workstation platforms on which our customers desire to run our applications. Cfront remains an important tool for us, especially for time-to-market considerations.

**Table 1: Module sizes**

module	classes	constructors	instr bytes	routines	<code>__sti__</code>
shlib_1	129	270	f472c	2068	109
shlib_2	656	1069	41dbc0	11513	329
applic_1	299	468	3b34e8	6205	207
applic_2	1123	1552	bcbf188	21593	577
applic_3	2129	2825	1f85fa8	32171	1301

The size of the average subroutine in Table 1 varies from 350 to 1000 bytes, and the ratio of total subroutines to `__sti__` routines varies from 20:1 to 40:1. Since `__sti__` routines are associated with compilation units and by default the instructions generated by the compiler for each

compilation unit occupy an interval of addresses, `__sti__` routines are separated by 7KBytes to 40KBytes on average. The virtual memory page size on many engineering workstations is 4KBytes or 8KBytes. Thus the probability of having more than one `__sti__` routine on a page is low. If starting a process involves demand paging the instructions from a disk that can service 50 page faults per second, then invoking an application will take several tens of seconds unless placement of `__sti__` routines is controlled to minimize page faulting. We observed this.

### Controlling placement of routines

Cfront cannot control the placement of `__sti__` routines because there is no way to talk about placement in the C language which is the output from cfront. A native C++ compiler can control placement by cooperating with the static binder to place together all code which performs initializations. Some compilation environments permit arbitrary programmer control over placement through the use of `#pragma` directives (or equivalent). In small software systems, it is conceivable for the programmer to control placement by putting each subroutine into a separate source file, so that the desired locality can be achieved by re-ordering the input list to the static binder.

We find it much more convenient to control placement of routines by using a post-load processing utility [JOHN90][LARU92]. A separate program takes as input a fully- or partially-bound module (application, shared library, dynamic module) together with a specification of the desired placement order, and then rewrites the module so that the subroutines occur in that order. Examples of such utility programs include MIPS Co.'s `cord` (cache order) and Hewlett-Packard's `fdp` (feedback-dependent positioning). We have also written our own utilities for platforms where they were not readily available. Such utilities are conceptually simple, although details such as span-dependent subroutine calling instructions [SZYM78] and maintaining correctness of debugging information can be hard.

In theory, it is also necessary to control placement of data (non-instructions); remember that the "hook" objects themselves (and possibly the `__link` structure, too) are touched during initialization. However, our applications have 10 to 20 times as many bytes of instructions as bytes of static data, so faulting of instruction pages matters much more to us.

Of course, a utility to re-order subroutines is useful for more than just reducing invocation time. We also apply re-ordering to reduce page faulting that would otherwise occur when processing interactive selection, menu picks, etc. We noticed that the working set for such computation tends to involve a couple member functions from one class, a few from another class, several from a third, etc. Although the total number of instructions is modest, the page faulting was not. Software development practice tends to place the implementation for each class in a separate source file. Since the working set of member functions from any one class tends to be sparse, the default ordering of subroutines in the virtual address space tends to increase page faulting. In order to improve interactive response, we find it necessary to reduce faulting of instruction pages by careful ordering of member functions across all classes. Determining what the order should be (which member functions are touched during menu pick?) is an interesting problem in itself. The usual profiling tools with clock-based sampling of the program counter tend to provide inadequate information. See **On-line visualization** for what we did.

### Reducing allocation hooks

Careful examination of the actual contents of `__sti__` routines reveals that allocation hooks account for the majority of initialization requirements; see Table 2 for one example. If all of the calls to the three allocation hook initializations are deleted, then 188 of the 295 `__sti__` routines disappear entirely. So what are these allocation hooks, why are they required, and why are they called hundreds of times?

Heading the list is our old friend, `Iostream_init`. Following are the hooks for initializing the

**Table 2: Calls from the 295 `__sti__` routines for one module**

calls	routine
282	[hook] <code>__ct__13Iostream_initFv</code>
218	[hook] <code>__ct__23Core_glb_strrgy_stk_iniFv</code>
180	[hook] <code>__ct__15Mule_value_initFv</code>
76	<code>__nw__FUi</code>
59	<code>__ct__4NameFPCci</code>
52	<code>__ct__7Ui_nameFPCc</code>
42	<code>__ct__9Core_uuid . . .</code>
41	<code>__ct__13Ddms_type_rep . . .</code>
38	<code>__ct__9Core_rstrFPCc</code>
29	<code>__ct__11Core_string . . .</code>
80	55 others (maximum 6 calls per routine)

global character-string registry (so that error messages can be printed from error status codes) and for insuring that predefined atoms (VOID, UNDEFINED, etc.) are properly connected with an interpreter. They are called so often because the corresponding `.h` header files are referenced (directly or indirectly) by most other source files. One of the corporate coding standards recommends that each class have a `dump()` routine which produces readable ASCII from an instance. One of the parameters to `dump()` is an `ostream_withassign`, defined in `iostream.h`. Thus `class.h` will `#include <iostream.h>`; therefore `class.c` (and any other file that references `class.h`) will get an allocation hook. Similarly, many classes have member functions that return status codes. In particular, many constructors return a status code through a reference parameter. Prior to the introduction of the exception facility [ARM90 ch15], the C++ language allows no way for a constructor to fail; yet the reality of programming is that construction CAN fail, and there must be a way to detect and recover. Few environments implement the experimental exception mechanism of C++, and even if they did there are still several mountains of code to convert. Dealing with status codes involves being prepared to print them, for which a character-string registry is necessary. So classes with status codes get an allocation hook to initialize the global stack of string registries. Any class that deals with the runtime interpreter gets an allocation hook to setup the global state of the interpreter.

These are three examples of a *subsystem*: a group of classes which cooperate to implement an API. If application invocation would recursively invoke the subsystems and initialize them bottom-up [as required by the idea of dependency on a subsystem], then the number of initializers would decrease from one per class [or pair of interacting classes] to one per subsystem [or pair of interacting subsystems]. Unfortunately, the capability for process subsystems is neither common nor popular among engineering workstations. In particular, System V Release 4 of the UNIX operating system specifies that shared libraries cannot be relied upon to implement subsystems: initialization order of shared libraries is explicitly allowed to be random [SVR4PG p 13-65]. Influential organizations interpreting C++ itself also display some unhelpful rigidity in this area: the *one definition*

*rule* (only one implementation of a given function is allowed per program [ARM90 p14]) and the *order of initialization rule* (lexical within a compilation unit, otherwise undefined [ARM90 p19]) should be applied within a subsystem only, but so far we have been unable to convince the powers that be. What a pity. When *divide and conquer* doesn't apply, then the programming calculus becomes particularly weak.

As a result of our experiences described above, we are trying to convert away from initialization by allocation hook. If initialization is necessary, then it should be implemented by explicit subroutine call. This is less safe (we might forget) and more tedious (copy-and-paste "by hand" the list of inits into the beginning of every application and subsystem), but it is the available, portable way to control initialization costs.

### **On-line visualization of program activity**

To aid in the diagnosis, understanding, and repair of locality problems that we encountered when using allocation hook initializers and object-oriented coding style in such large programs, we developed a facility for real-time address tracing, with on-line interactive graphical visualization. It is fast enough to analyze our interactive applications in real time. The facility is based on a post-load program processor [JOHN90] (also known as "exec editor" [LARU92], or a.out re-writer) and takes advantage of the SPARC International application binary interface (ABI) [SPARC92] and the X Window System display system. Although the current implementation is specific to the SPARC architecture, the results and insights obtained are independent of platform across operating systems that employ similar models for process address space. In particular, we apply the understanding that we gain from analyzing execution under Sun Microsystems' SunOS 4.x operating system to all platforms on which our applications run. Although the visualization tool might seem irrelevant to the conclusions regarding static initializers, sensory sight was integral to understanding and appreciating what actually happens. Sight is also an extremely effective medium for convincing programming managers what the problems are, and what can be done about them.

The post-load processor reads a module, disassembles it instruction-by-instruction, then writes a new module containing additional code which intercepts memory load/store and subroutine entry/exit. At execution, the intercepted data is buffered before being processed and displayed. The volume of intercepted data is reduced by recording only subroutine entry/exit for code, and by ignoring data references made with a constant offset into the current stack frame. Our subroutines are small enough that basic blocks have little effect on page fault behavior, and the utility programs that actually re-order code work subroutine-by-subroutine anyway. Stack usage is small and well-understood from prior measurement and analysis. Besides, the stack is visible through uses of automatic local variables passed by reference, and through local arrays.

The two linear sequences of subroutine ordinals and data memory addresses are mapped into separate two-dimensional display regions. The mapping uses a Hilbert space-filling curve [HILB91] [WITT83] [COLE83] [FISH86] [GRIF86] to provide good perceptual communication of locality, even at high data rates. Two points that are near each other in the linear sequence will be near each other on the display, and two points that are far apart on the display are also far apart in the linear sequence. This is as good as one can do. In particular, there are points that are near one another on the display, and far apart in the linear sequence. Points on the display are actually small filled squares of pixels. If a point has been touched in the most recent batch, then its color is white. As time and batches of intercepted data pass without a point being touched, then its display decays through a colormap-animated temperature spectrum, finally turning ash gray to signify that it was touched a long time ago.

In the code display, selection (Button1 press-drag-release) outlines a rectangular area. Activation (Button2) alters the display by moving the recently-active (non-gray) points from within the rect-

angle to the front of the linear order, and appends the names of the corresponding subroutines to an output file. Future display updates show the effect as if the subroutines had been physically re-ordered, which can be achieved by feeding the list in the output file to the re-ordering utility.

In the data display, selection outlines a rectangular area and fills the minimum linear address-space interval that covers all the points within the rectangle. (This provides feedback on the Hilbert curve mapping.) Activation creates a new window, data x code, that analyzes future memory references. The horizontal axis is the address-space interval that covers the outlined rectangle, and the vertical axis is occupied by subroutines that reference the interval on a first-come-first-recorded basis. Periodically, the routines on the vertical axis are sifted so that the ones touching the most pages appear towards the top. In the data x code display, a full-window crosshair tracks selection. The lower left corner contains the name of the subroutine that corresponds to the horizontal arm of the crosshair.

When rewriting the original module, the post-load processor uses one or more of the SPARC "global" (not register-stack windowed) registers to reduce execution time. The SPARC ABI reserves registers %g2, %g3, and %g4 to applications, and registers %g5, %g6, and %g7 to the system. However, common compilers pre-date the ABI, so by observation %g6 and %g7 are the only registers that are actually unused. A one-word (32-bit) datum can be intercepted and recorded in 10 instruction times, 15 if re-entrancy is required. Since memory references occur about 1 instruction in 5, the slowdown due to interception and recording is a factor of 2.8 or 3.8. Analyzing and displaying the results also take time, although a separate processor can be used for the X window system server. In such cases, users complain once in a while that the display is updated too fast to process by eye-ball and understand in real time.

Figure 2 through Figure 5 attempt to show what the visualization tool displays. Although captured from real execution, the images have been processed to deal with limitations in printing. The spectrum has been reduced to 4 steps from a typical 16, black-and-white has replaced color, and the entire image appears in reverse video so that white predominates.

### Availability

The visualization software is not available for general distribution; contact the author:

### References

- [ARM90] M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, (Reading, MA: Addison-Wesley) 1990; ISBN 0-201-51459-1.
- [COLE83] A.J. Cole, "A Note on Space Filling Curves," *Software Practice and Experience* **13.12** 1181-1190 (December 1983).
- [FISH86] A.J. Fisher, "A New Algorithm for Generating Hilbert Curves," *Software Practice and Experience* **16.1** 5-12 (January 1986).
- [GRIF86] J.G. Griffiths, "An Algorithm for Displaying a Class of Space-filling Curves," *Software Practice and Experience* **16.5** 403-412 (May 1986).
- [HILB91] D. Hilbert, "Über steige Abbildung einer Linie auf ein Flächenstück," *Math. Annln* **38**, 459-460 (1891).
- [JOHN90] S.C. Johnson, "Postloading for Fun and Profit," *Proc. Winter 1990 USENIX Conf.* (Washington, DC: January 22-26, 1990) 325-330.
- [LARU92] J.R. Larus and T. Ball, "Rewriting Executable Files to Measure Program Behavior," University of Wisconsin Computer Sciences Technical Report 1083 (Madison, Wis-



consin: March 25, 1992); [larus@cs.wisc.edu](mailto:larus@cs.wisc.edu) .

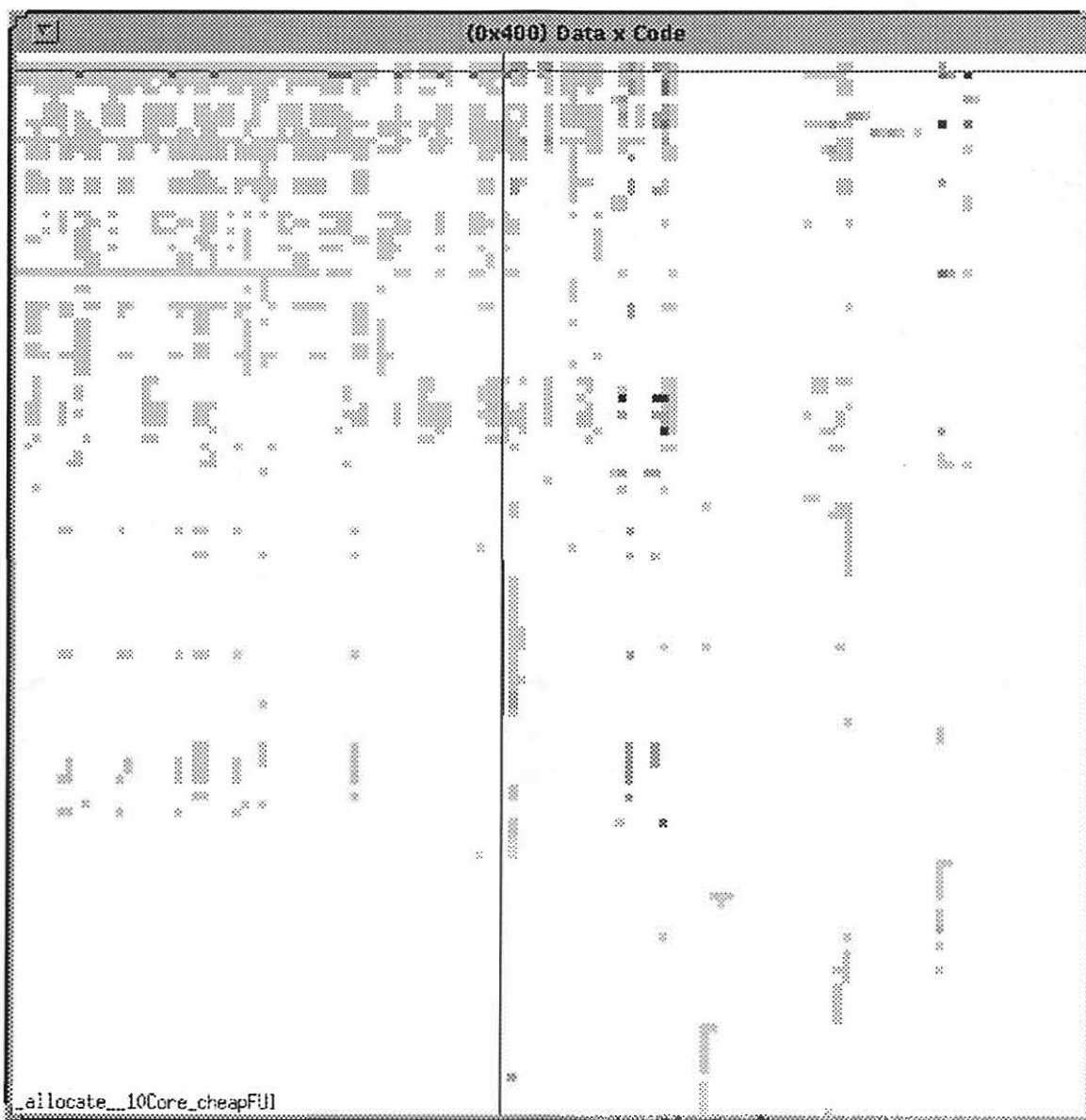
[SPARC92] SPARC International, **The SPARC Architecture Manual**, (Englewood Cliffs, NJ: Prentice-Hall) document SAV080SI9106 Version 8, 1992; ISBN 0-13-825001-4.

[SVR4PG] UNIX System Laboratories Inc., **UNIX System V/386 Release 4 Programmer's Guide: ANSI C and Programming Support Tools**, part number 430415 (1991).

[SZYM78] T.G. Szymanski, "Assembling code for machines with span-dependent instructions," *CACM* 24.4 (April 1978).

[WITT83] I.H. Witten and B. Wyvill, "On the Generation and Use of Space-filling Curves," *Software Practice and Experience* 13.6 519-526 (June 1983).

Figure 2. Data x Code display. Pages touched by a subroutine appear in a horizontal row.



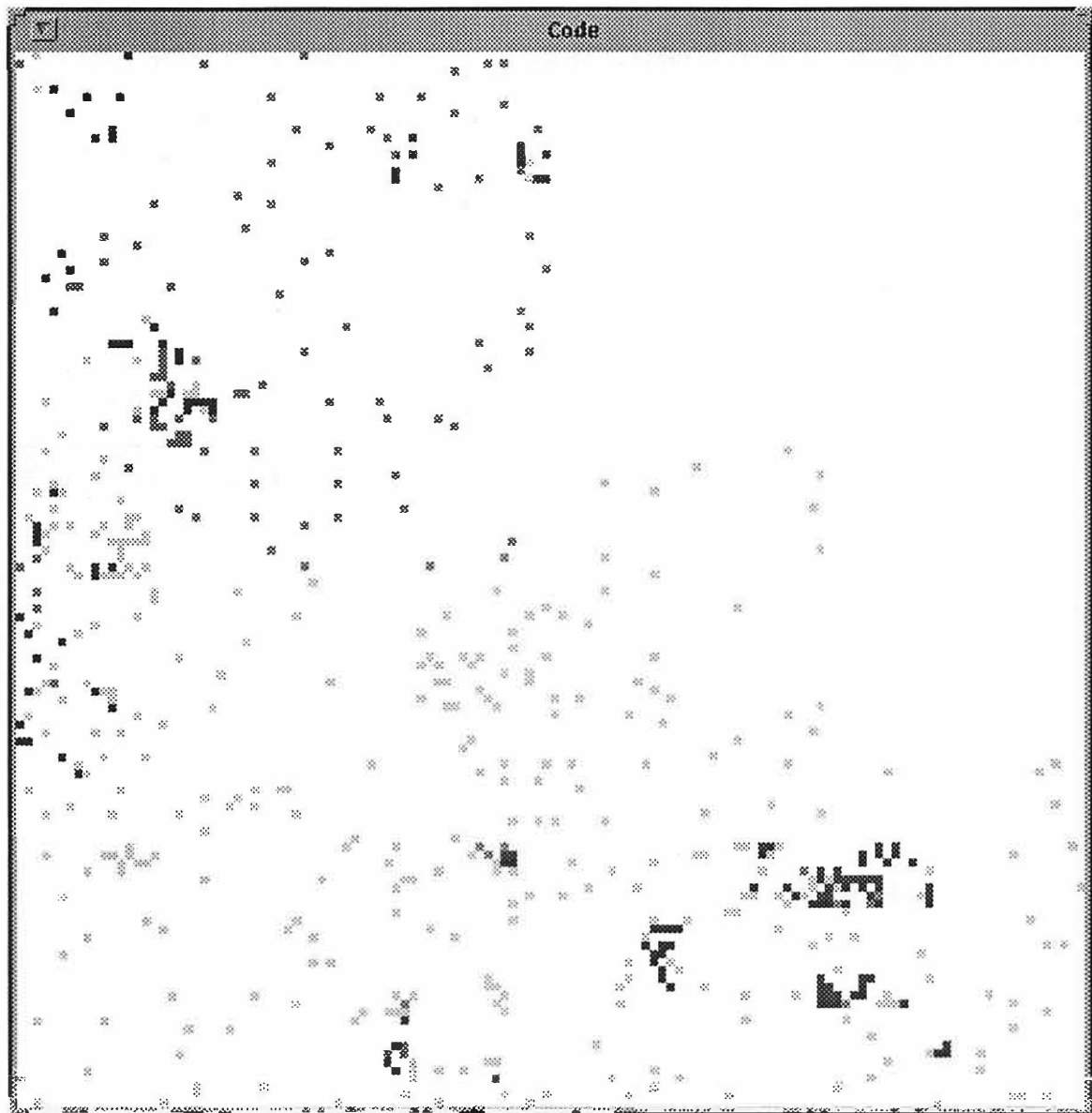


Figure 3. Code display with default uncontrolled placement of `__sti__` routines. The window contains space for 16384 subroutines; 13802 subroutines are present.

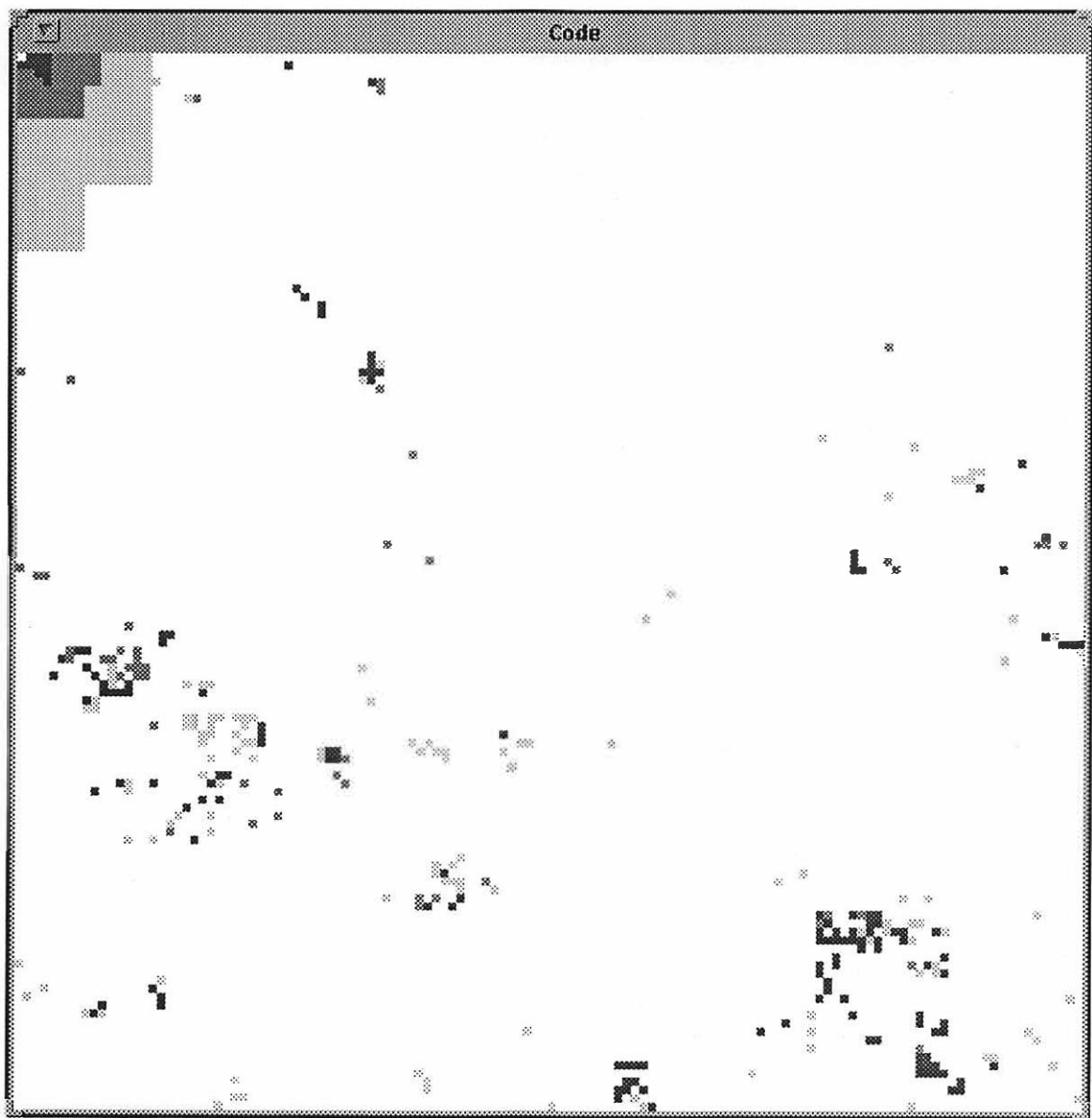


Figure 4. Code display re-ordered with `__sti__` routines grouped together.

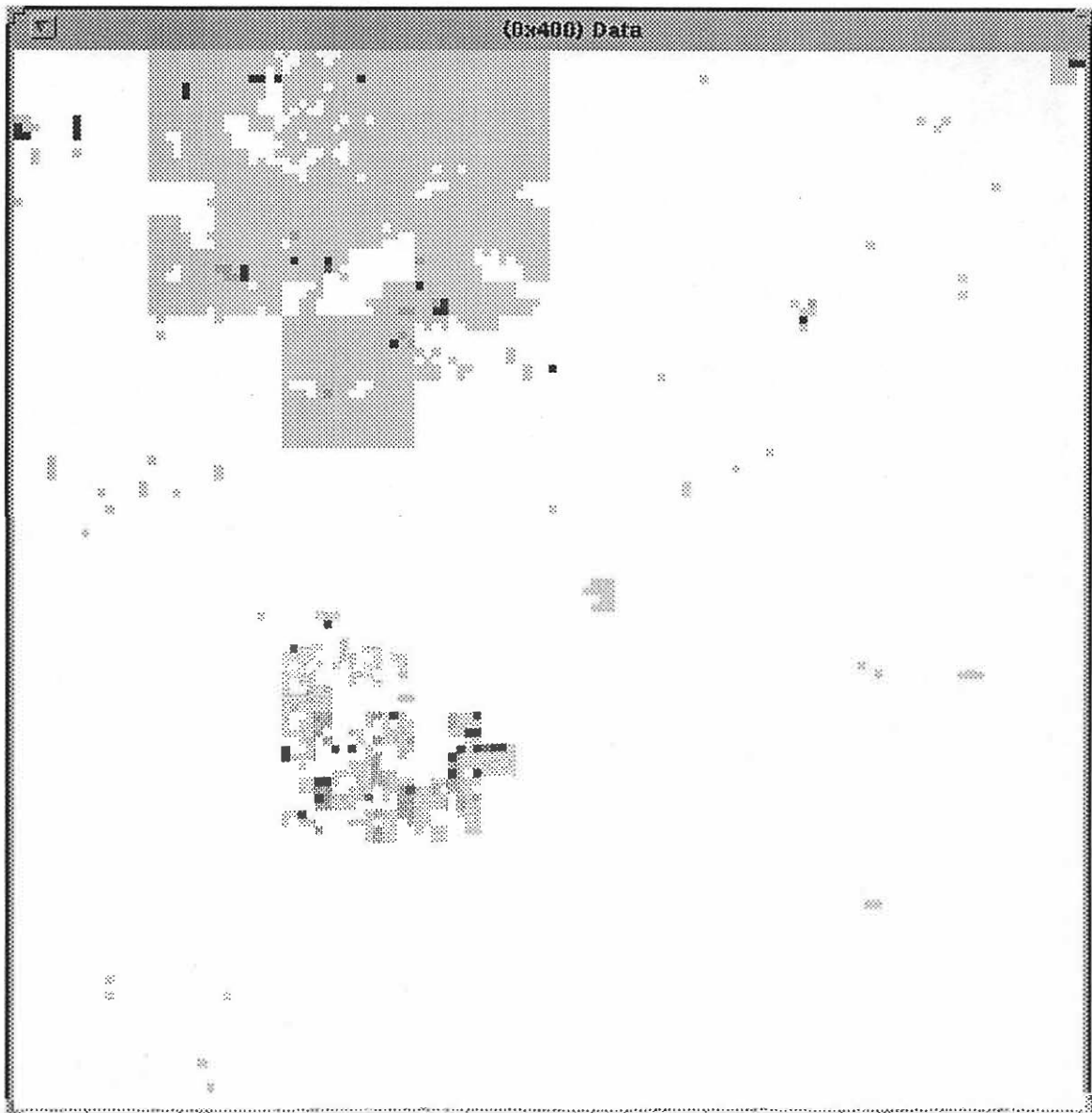


Figure 5. Data display. 0x400 is the page size. 16384 pages (the bottom 24 bits of the address space) are displayed. The stack is in the upper right corner. The large gray area in the upper left quadrant is the heap; its size is approximately  $7 \times (1/4 \text{ megabyte})$ . The smaller area in the lower left quadrant is the static data area of a shared library. Pox are other shared libraries, or jump tables.

# Cdiff: A Syntax Directed Differencer for C++ Programs

Judith E. Grass  
AT&T Bell Laboratories  
600 Mountain Ave., Rm. 3c532c  
Murray Hill, NJ 07974  
grass@ulysses.att.com

## ABSTRACT

*Cdiff* is a program that detects syntactically significant changes between different versions of C++ programs. *Cdiff* is an application implemented within the framework of C++ *Information Abtractor* system and the newest member of the CIA++ toolkit. Because it is built upon this system, *Cdiff* can support complex queries about the differences between versions.

This paper discusses why *Cdiff* is a useful addition to a C++ programmer's toolkit and its implementation. The paper contains examples of *Cdiff* queries that show how *Cdiff* queries make analyzing implementation changes easier than is possible using common textual differencing tools (i.e. *diff*). Two different releases of the *lib/InterViews* library from *InterViews* 3.0 are used as a test case.

## 1 Introduction

*Cdiff* is a tool that supports the analysis of changes between different versions of a C++ program. Unlike familiar textual tools, such as *diff*, *Cdiff* reports only syntactically significant changes in a C++ program. Differences in whitespace or comments appear only on the file level in a *Cdiff* analysis. The do not affect the analysis of other program units.

The *Cdiff* tool is built as an application in the CIA++ (the C++ *Information Abtractor*) toolkit [5, 4]. The CIA++ system generates a database of static analysis information for a C++ program. The database supports a variety of program analysis tools. Each tool reuses the information in the database, so tools do not need to parse C++ programs or to maintain their own analysis data. The CIA++ database optionally generates checksums for every C++ symbol, file or macro. The checksums make it possible to quickly identify entities that have been added, deleted or changed between versions of a program. The CIA++ system is briefly described below. *Cdiff* is the first tool based on CIA++ to do analysis across two programs.

The ability of the CIA++ toolkit to deal with complex queries makes it possible for *Cdiff* to support very specialized queries about how a program has changed. For instance, *Cdiff* can generate a list containing only the changes in public member functions. This is particularly useful for investigating interface changes in libraries. Queries based on function signatures, symbol names or membership in specific classes are also possible. This paper will be illustrated by queries examining two versions of the *InterViews lib/InterViews* library.

The *InterViews* package is a C++ graphical interface toolkit developed at Stanford University [6, 7, 8]. I have chosen the *InterViews* library as a test case for several reasons: it is complex; it is familiar to many people programming in C++; and it is publicly available. Moreover, I have used the *InterViews* library as a test case for the other tools in the CIA++ toolkit [4]. The sample *Cdiff* queries are based on databases generated for the *lib/InterViews* library (for the *InterViews 3.0beta* release and the latest *InterViews* release, 3.0.1). The entire *InterViews* package contains several additional libraries, but including these would unnecessarily complicate the analysis. Two versions of just this library are different enough to make the query results interesting.

## 2 Motivation

*Cdiff* is a child of necessity. The front end of the CIA++ system, *ciafront*, is embedded in a copy of the source code for *cfrent*. In maintaining the code for *ciafront*, I have had to track the changes

in cfront and merge those changes into my own program. So long as the changes I tracked were based only on the *beta* releases of cfront 2.0 and cfront 2.1, the task was manageable using nothing beyond the familiar tool, *diff*. File structure did not change between versions, and the ratio of noise to significant textual change was relatively small.

My first encounter with the source for cfront 3.0 made it abundantly clear that using this approach for upgrading cfront to cfront 3.0 was a fool's errand. Running *diff* on obvious file pairs generated over 28 thousand lines of textual differences. This did not include the new files introduced in cfront 3.0 and it did not begin to indicate which definitions these changes affected.

The list of symbol differences generated by the present conservative prototype of Cdiff are an order of magnitude smaller. It indicates about 2500 symbols that have been added, deleted or changed. This list is both specific to the things defined in the program and of a manageable size. The list of differences generated by Cdiff also contains many fewer "red herrings" than the output of *diff*.

The toolkit for *CIA* (the *C Information Abstractor*) [1, 2]<sup>1</sup> has an older syntactic differencing tool, *ciadiff*. This tool does not support generalized queries, and it most definitely does not support C++. The original implementation of Cdiff was quite similar to *ciadiff*. The additional complexity of the C++ language forced me to deal with many more issues in Cdiff, and so the tool rapidly evolved away from its origins.

### 3 An Introduction to the CIA++ System

It is impossible to fully understand the flexibility or nature of Cdiff without some understanding of the CIA++ system. That system both does the analysis that allows changing symbol definitions to be detected and supports the query and display facilities that are essential to the implementation of Cdiff.

The CIA++ system is made up of three major components: *ciafront*, which analyzes a C++ source module and stores the results of the analysis in a database module; the database linker, which combines the modules into a merged relational database; and the *ShareView* tools, which access the database to answer specific queries about the program. Cdiff is one of these tools. The database schema and basic toolkit are described in detail in [5]. A different paper gives an extended example of using these tools for design analysis [4]. This section provides only a synopsis of that information.

#### 3.1 Ciafront

*Ciafront* scans, parses and generates a single database module for a single C++ file. The database generated includes data from the root file as well as from all included header files. The database contains information about five kinds of C++ entities: files, macros, types, functions and variables. Members of classes are included. Entities declared within functions are not recorded. The CIA++ database also saves many kinds of cross-referencing information. The cross-referencing capabilities are largely irrelevant to the Cdiff program.

A complete database for an entire C++ program is built by linking together all of the individual database modules generated from the C++ source files.

#### 3.2 The Basic CIA++ Tools

The core of the toolkit is made up of a small set of packaged query commands: *Def*, *Ref*, *Viewdef*, *Viewref*. Only the *Def* query is directly relevant to the Cdiff tool.

<sup>1</sup>CIA is a precursor of CIA++. Both CIA and CIA++ are actively used in AT&T.

```

falcon 29> Def type ivCanvas
file          dtype          name          bline df
=====
/InterViews/canvas.h class      ivCanvas      47    df
e/InterViews/glyph.h class      ivCanvas      39    dc
/InterViews/window.h class      ivCanvas      38    dc

falcon 30> Def -u type ivCanvas
736;ivCanvas;t;/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/include
/InterViews/canvas.h;class; ;47;0;115;df; ; ; ;b65dca1a
736;ivCanvas;t;/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/include
/InterViews/glyph.h;class; ;39;0;39;dc; ; ; ;22a2185f
736;ivCanvas;t;/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/include
/InterViews/window.h;class; ;38;0;38;dc; ; ; ;22a2185f

```

Figure 1: A simple definition query

Def executes basic queries about entity declarations and definitions. For example, the query `Def type ivCanvas` extracts database information about the type named *ivCanvas*<sup>2</sup>.

The formatted response to this query appears in the top half of figure 1. The bottom half of that figure contains an example of the unformatted output for the same query. The formatted query shows the name of the type *ivCanvas*, the truncated name of the file in which it is declared, its declared type *class*, the line that the declaration began on and the fact that this declaration is actually a definition (*df*)<sup>3</sup>. The class *ivCanvas* has additional declarations in the header files *glyph.h* and *window.h*, as shown by the lines marked *dc*.

The raw, unformatted version contains complete information that is not truncated, but neither is it easy to read. All formatted displays are implemented by generating an unformatted response and passing it through a formatting filter. The raw form is intended to be used as input to other analysis, formatting and user interface tools, such as Cdiff. This makes it possible for any user to expand the toolkit or to customize the displays using little more than shell or awk scripts.

The final hexadecimal number appearing in the raw output line is especially important to the Cdiff application. That number is a checksum generated by hashing together all of the syntactically relevant components that contribute to the definition (or declaration) of *ivCanvas*. From this alone we can see that the declaration of *ivCanvas* that appears in the file *glyph.h* is the same as the declaration that appears in *window.h*. The generation of checksums is discussed below.

The basic CIA++ Def query has the syntax:

```
Def Entity_description Optional.Selectors,
```

where *Entity\_description* includes the specification of an entity kind: *file*, *func*, *var*, *macro*, *type* or *-* and a name. The symbol *-* is a *wild-card* that matches all names and all entity kinds. The *Optional\_Selectors* can be used to restrict the search to database entries with specific characteristics. Regular expressions may be used for names and may occur in the selection clauses. For example, the command

```
Def type - dtype="~class$|^struct$" df="df"
```

<sup>2</sup>In InterViews 3.0 there is a convention that most class names from the 3.0 version of the library carry an *iv* prefix. This is not obvious from a casual reading of a source code file. Each significant class name in the source code is, in fact, a call to a macro that expands the name. In this example, *ivCanvas* appears in the source text as *Canvas*.

<sup>3</sup>Technically, these class "definitions" should be referred to as "complete declarations", since the ARM [3] says that "definitions" must allocate memory. In the case of CIA++, the definition flag field is used to find the fullest possible specification of all macro, type, data and function entries. For classes, this is the declaration that contains all the member declarations and inheritance information. It is useful for these to be covered by one abbreviation.

extracts a list of all types defined as either `class` or `struct`. The word *dtype* is an abbreviation for “data type”. The final selection clause confines the output to only actual data definitions, as the abbreviation *df* stands for “definition flag”. The notation used for these commands is terse and rather opaque. The intention is to build interfaces on top of these and hide them from naive users.

Other tools in the CIA++ toolkit present information about cross-referencing and generate maps of these relationships. These include call graph maps, inheritance maps and file inclusion graphs. Other tools generate statistics about component connectivity (*Ciafan*), locate unused entities (*Deadcode*) and find strongly connected subsystems (*Subsys*).

## 4 Cdiff: A Syntactic Differencer

There are several tools in common use to track differences between versions of a program. The *diff* tool is the most familiar of these. Other tools are intended to be used primarily on code. These include “stand-alone” programs like *ncsldiff* and programs that are part of change management systems, like *rcsdiff* in the RCS system [9]. In all of these cases, comparisons are made textually. These programs cannot parse programs and have no structural information to guide their actions. Using these tools uncovers the files that have changed, but only provides a weak and unstructured view of the nature of the changes.

Commonly, users must cross reference the reported textual changes with the original and changed files to understand the significance of the changes to the structure of the program. This is a time consuming and error prone activity. The use of multi-windowed environments makes the cross checking easier, but this is still a labor-intensive process.

Cdiff allows changes in specific collections of symbols to be studied. The changes it reports are structural, and relate directly to how the program performs. Since the tool is based on a full syntactic and static semantic analysis of a program, Cdiff can isolate changes in a file’s compilation that result from changes in included files and from changes in compilation options.

### 4.1 Building the Databases

The Cdiff tool depends on the existence of a CIA++ database for each program being compared. Under normal software development conditions these databases would be generated as the program was being developed. Older versions of the database could be saved using change management systems like RCS. In looking at different versions of InterViews, the conditions are somewhat different. This is not code that is locally developed.

The process of building CIA++ databases for a version of InterViews is essentially the same as compiling the code to install it. With some editing, the generation of the databases can be put under the control of the InterViews *Imake* files. In general, the file dependencies and compilation options used to build a CIA++ database are the exact same dependencies and options used to compile the code. Often all that must be done is substitute the `CC` command with a `CIA` command. For checksumming an additional `+X` option must be added.

It is even easier to use *nmake* to generate databases, because *nmake* has built-in CIA++ rules. If an *nmake* specification file contains enough information to build a C++ program, then it can also automatically build the CIA++ database for that program without any additional information. For InterViews, it is especially convenient to use *nmake* to build the databases. On my system, the source code for each of these two releases of InterViews is installed in separate system directories that have extremely long, and totally disjoint path names. Since I do not have the luxury of generating the databases in system space, I must build the databases in my local space. This is the command that must be typed to build one module of the database for release 3.0.1 in a local directory, if done entirely by hand:

```
falcon 33> CIA +X -I/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src\  
/include -c /usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src\  
/lib/InterViews/hit.c
```



```

Added files:
  In directory: /usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/lib
/InterViews
    aggr.c                187 lines
    arrcomp.c             47 lines

Changed files:
  In directory: /usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/lib
/InterViews
    align.c               107=>108 lines
    background.c          74=>72 lines
    border.c              101=>97 lines
    box.c                 421=>428 lines
    center.c              107=>108 lines
    character.c           105=>94 lines
    composition.c         745=>810 lines
    compositor.c          39=>42 lines

Deleted files:
  In directory: /n/gryphon/g2/InterViews/iv/src/lib/InterViews
    aggregate.c           189 lines
    arraycomp.c           45 lines
    box2_6.c              481 lines
    brush.c               90 lines
    color.c               131 lines

```

Figure 2: Reported file differences

The `-c` option indicates that a `.A` database should be generated without generating a full database. This command is roughly equivalent to the command for generating a `hit.o` file in the same directory. In that case the command would be `CC` and the `+X` option would be dropped. The length of the path names is a burden that appropriate automation can ease. Using `nmake`, the entire InterViews 3.0.1 database can be built by typing:

```
nmake CIADB
```

in some directory. The database for the 3.0*beta* release must be built in a different directory using another `nmake` specification. For purposes of demonstration, a databases for 3.0.1 was built in `IVDB/IV3.0` and for the *beta* version in `IVDB/IV3.B`. Once databases have been constructed, `Cdiff` can be used to explore differences. Currently both `CIA++` and `CIA` expect a directory to contain at most a single database<sup>4</sup>. So, although it is possible to run `Cdiff` queries from an arbitrary directory, the location of both databases must be specified. This can be done within the context of a shell process by exporting two environment variables (`$OLDCIADIR` and `$NEWCIADIR`), or through options to the `Cdiff` command.

For applications that have not changed directory structure or without multiple files sharing the same name, this is all the preparation that is needed to run `Cdiff` queries. InterViews, as it is represented on my system, meets neither of these criteria. In order to get accurate results from the queries, the `Cdiff` tool must be supplied with information about directory equivalences. Either a shell environment variable or a command line option may be used for this. For InterViews, the

<sup>4</sup>This is due to limitations imposed by the database management system used. Many DBMS have similar restrictions.

long names make the path equivalencies truly hideous. involved in determining file equivalency. A convenient way of setting the environment for complex systems like InterViews is to write an initialization shell script. The appendix contains InterView's initialization shell script. The issues involved in the determination of file equivalence are discussed below.

Once the environment has been initialized, queries can be run. This query:

```
Cdiff file -
```

identifies all of the files that changed between releases 3.0*beta* and 3.0.1 of InterViews.

The response to this query is too large to fully present in the body of this paper. It indicates that 68 files have been deleted, 22 added and 108 changed between these versions. The size of the response to this unrestricted query suggests that there are substantial differences.

Arbitrary slices of the file changes can be extracted by replacing the wildcard symbol ('-') in the above query by *egrep*-style regular expression describing a group of file names. This query:

```
Cdiff file ".*/[a-c][^/]*.c$"
```

extracts only the differences affecting source files with names that start with the letter 'a', 'b' or 'c'. The complete output for this query is presented in figure 2. Although this sample query is somewhat artificial, it demonstrates some of the advantages of power and expressiveness that Cdiff has over the traditional textual differencing tools.

The format shown in figure 2 is used only to display file changes. In this format the file changes are grouped first by the kind of change (addition, deletion, change), then by directory. The additional *lines* notation refers to the size of the file. For files that have been changed, both the old and new line count are presented. Changes are detected in both the source code files and all of the included header files.

All formatted output is generated by applying filters to the raw output generated by the differencing engine. A small sample of the raw, unformatted output is presented in figure 3<sup>5</sup>. The raw output contains a lot of additional information that could be used by other customized presentation filters and other applications.

The first field of the raw output indicates whether the file named is from the old program ('<') or from the new one ('>'). Whenever possible, changes are labelled with the name of the new file. The second field, *c*, indicates that the entity has *changed*. The letter *f* in the third field indicates that the entity is a file. The field following that is the directory containing the file. The name of the file is given in the fifth field. The hexadecimal number is the checksum generated for the original version of the file. The entries that follow the checksum are line numbers. The other fields in the raw output are not meaningful for file entries, and all but the first three fields and the line count information would be suppressed by formatting. However, the assignment of field positions is consistent for all entity kinds.

It is possible to get a listing of *everything* that changed in these two versions with the following command:

```
Cdiff - -
```

where a wild-card is used both for the kind of the entity and its name. In light of the number of changes suggested by the file changes, the result will be large and unwieldy. More specific queries can be written to focus on specific files or specific symbols. The following query isolates the macro changes that occur in the file *regezp.c*:

```
Cdiff macro - file=regezp.c
```

---

<sup>5</sup>In fact, the so-called "unformatted" output is rigidly formatted. Every field in it has an assigned meaning and a known range of values. It is only unformatted in the sense that it has not been laid out in a way that is convenient for human readers.

```
>;a;f;/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/lib/InterViews;aggr.c; ;
/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/lib/InterViews/aggr.c; ;df; ;
fb4563eb;1;187; ;
<;d;f;/n/gryphon/g2/InterViews/iv/src/lib/InterViews;aggregate.c; ;/n/gryphon/g2/Inter
Views/iv/src/lib/InterViews/aggregate.c; ;df; ;1a41a3c;1;189; ;
>;c;f;/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/lib/InterViews;align.c; ;
/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/lib/InterViews/align.c; ;df; ;
df77c13e;1;107;1;108; ;
```

Figure 3: A sample of unformatted output

Changes in file /usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/lib/InterViews  
/regexp.c

```
349-349 d      mac CHARBITS
348-348 d      mref CHARBITS
a 349-349      mac RE_CHARBITS
a 348-348      mref RE_CHARBITS
352-352 c 352-352 mac UCHARAT
```

Figure 4: Macros that have changed in file *regexp.c*

and produces the complete results shown in figure 4. The marking *mac* means that the name that follows is a macro definition. The notation *mref* marks a reference to an otherwise undefined macro. Usually references are generated by *ifdef* preprocessor statements. The line numbers refer to the beginning and ending line of the definition or reference. For deletions, indicated by the letter 'd', they are the lines in the old file. For additions, indicated by 'a', they are lines in the new file. Changes, indicated by 'c', show the lines in the old file followed by the lines in the new file. This coding and line numbering convention is used for reporting all changes except file changes.

The output indicates that a macro *CHARBITS* was deleted and another macro *RE\_CHARBITS* was added. Probably what happened was that *CHARBITS* was renamed, but Cdiff does not attempt to draw such inferences. In general, changes in a name cause a change in the checksum, so there is no basis to infer that these share a definition other than the similarity in names and line numbers.

Since InterViews claims to be an object-oriented program, a logical place to start an investigation of differences is with changes in the class structure. A complete list of classes that have changed can be extracted with the query:

```
Cdiff type - dtype="~class$|^struct$" df="df"
```

where *dtype* is a regular expression describing the data type of the items sought. The expression *df=df* restricts the search to changed definitions only. Otherwise, changed simple declarations would also be detected.

The results of this query shows 78 added, 99 deleted and 82 changed classes. Since a CIA++ Def query on InterViews 3.0.1 shows 231 classes in all, this is a substantial amount of change. The query:

```
Cdiff type .*Window dtype="~class$|^struct$" df="df"
```

shows a slice of these differences affecting classes containing the word "Window" in their name (figure 5). Note that the disappearance of the definition of *ivInteractorWindow* does not necessarily mean that that entity is gone. It means that it needs to be investigated further. If you make a few

Changes in file /n/gryphon/g2/InterViews/iv/src/include/InterViews/2.6/InterViews/iwindow.h

```
38-63 d      type class ivInteractorWindow
```

Changes in file /usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/include/InterViews/window.h

```
114-157 c 117-169      type class ivManagedWindow
170-179 c 182-191      type class ivTopLevelWindow
181-191 c 193-203      type class ivTransientWindow
a 46-113      type class ivWindow
```

Figure 5: Window class definitions that have changed

Changes in file /usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/lib/InterViews/tile.c

```
70-126 d void ivTile::allocate (ivAllocation&,int,ivRequisition*,ivAllocation*)
a 72-127 void ivTile::allocate (const ivAllocation&,ivGlyphIndex,const ivRequisition*,ivAllocation*)
34-68 d void ivTile::request (int,ivRequisition*,ivRequisition&)
a 34-70 void ivTile::request (ivGlyphIndex,const ivRequisition*,ivRequisition&)
```

Figure 6: Changes in the public interface of *Tile*.

queries on the 3.0.1 database, you will find that there are some remaining declarations and references to *ivInteractorWindow*. Remember that these databases contain only one large subcomponent of InterViews, the *lib/InterViews* library. Disappearing definitions of classes and functions may simply reflect a reorganization of the entire package's directory structure. If the databases contained information from either a complete program or the entire InterViews package, the deletion of a definition would unambiguously imply the deletion of the entity.

It is possible to pull out complete lists of modified variables and functions (including all of the class members) with queries like the ones we have shown. A better strategy is to investigate the changes in interesting classes independently. Beyond that, the aspects of change that a Cdiff user would focus on depends on the user's need. An InterViews library user would probably confine the investigation to the public interfaces of the classes used in a particular application. A InterViews developer would dive into the private and protected parts of the classes she (or he) was maintaining.

A library user with code depending on the *ivTile* class would want to determine if the interface to that class had changed. The following query would reveal any public member changes (for data, functions or types, since '-' is the wildcard):

```
Cdiff - ivTile::.* df=df ms=pb
```

where *ivTile::.\** is a regular expression selecting all the members of *ivTile* and *ms=pb* indicates that only items whose *Member\_Scope* is public should be chosen. The results of this query are in figure 6.

If the user's code happened to depend on the versions of the functions *ivTile::allocate* and *ivTile::request* that have disappeared, it would be necessary to investigate further to see if these represented simple interface changes, or if the semantics of these functions had also changed. Since changes in interfaces generally imply changes in the names of formal arguments, it does not seem

```
Changes in file /usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/include/  
InterViews/color.h
```

```
37-37 d      const int ivColor::Alpha  
a 43-43      const int ivColor::Invisible  
37-37 d      const int ivColor::Nop
```

Figure 7: An sample list of changed variable definitions

possible to generate checksums in a way that would distinguish these two cases. Notice that a function whose body has changed without changing the interface will only show up as a *change* entry. The change may still be significant to a user, but determining that is a deeper kind of analysis than any static analysis tool can do.

There are few public data members in InterViews code, and only a handful of those have changed between these versions. For the sake of completeness, I am including this query:

```
Cdiff var ivColor:: ms=pb
```

which extracts the changes in public data members for the class *ivColor*. The results appear in figure 7. All of these constants are enumerator values.

Unlike traditional textual differencing tools, Cdiff allows change to be analyzed on the basis of an entire program, rather than through the fragmented view obtained by file-to-file comparisons. It allows great flexibility in extracting changes in particular meaningful program components and it organizes and presents the results in ways that are meaningful for the structure of C++ programs.

There are limitations. Cdiff detects and reports all of the changes it finds, but it is not capable of extracting the deeper semantic meaning of those changes. At this time only a human being is capable of making those judgements. However, Cdiff can help immensely in locating the program structures that must be studied.

## 5 Implementation

Cdiff is implemented as part of the CIA++ toolkit. It depends on the checksums contained in a CIA++ database and the definition queries the system provides. Cdiff proper is written using a Korn Shell driver and a C++ program that compares database entries. The output formatter is a simple AWK program.

### 5.1 Generating Checksums

Checksums are optionally generated for every entity declaration and definition that appears in the database. For variables, functions and types, checksums are generated during parsing. File checksums and macro checksums are generated independently.

The C preprocessor used with *ciafront*<sup>6</sup> passes information about macros to *ciafront*. This information includes a checksum for every macro definition. *Ciafront* accepts the value, enters that into the database, and asks no further questions.

Changes in macro definitions sometimes affect the checksums generated for other symbols. Merely changing the name of a macro will not change the checksum of any variable, type or function that refers to it. These checksums are generated long after macros have been expanded. If changing the definition of a macro results in a syntactically significant change to the code, it will change the checksums of entities that refer to that macro. Macros changes that do not generate differences in code do not cause changes in these checksums.

---

<sup>6</sup>This is a preprocessor developed at Bell Labs by Glenn Fowler. His preprocessor and preprocessor library make it possible to track macro information in the CIA++ databases.

Practically, this is about all that can be done. Macros have effects that cannot be bounded in scope, delimited or reversed. Cross referencing between macros and C++ programming objects is, for semantic reasons, cannot reliably be maintained. This makes macros a significant headache for tool builders. As a result, it is desirable to report significant changes in entities that refer to changed macros. Otherwise it would be difficult to identify the cascading effects of such a change.

File checksums are currently generating by hashing on all the lines in a particular source or header file without consideration of preprocessing. This means that, for files and files only, changes in whitespace and changes in comments are reflected as changes in the checksum. Whether or not this is desirable depends on the situation. In it's favor, this kind of checksumming allows a file's checksum to be independent of the checksums of the files that it includes. In addition, there are cases in which changes in whitespace and comments need to be considered. On the other hand, in some cases this is noise. It is also possible to generate a file hash by combining information from all of the definitions contained within the file. This alternative may be added in the future.

Checksums for variables, types and functions are generated during parsing. Each token is hashed as it is recognized by the lexer. Each production returns a function of the hash values for its tokens as a synthesized attribute. If the production declares or defines a symbol, the accumulated checksum of the syntax tree for the body of the declaration is assigned to that symbol. Lexical and syntactic changes that are significant to the parser show up as changes in the checksums of symbols.

This approach gives a somewhat conservative picture of the syntactic changes in a program. There are some kinds of modifications that will show up as significant changes that, in fact, have no semantic effect on a program. In general, renaming a local variable has no effect, so long as the new name does not collide with an existing name. Such changes show up in the CIA++ database as a checksum change in the function's database entry. At the very least, isolating these kind of pseudo-changes would require some kind of dataflow analysis. This is beyond the scope of our tools.

It is theoretically possible that some symbols could checksum to the same value. This can only result in masking a modification if the checksum for a particular symbol remained unchanged. Experience with the hash package used for checksumming indicates that this is extremely unlikely to happen in practice.

## 5.2 Generating Change Lists Within Cdiff

Cdiff queries have essentially the same syntax and semantics as the CIA++ toolkit's Def query. This is not an accident. The Cdiff shell script works by running a Def query on the old and new program database and then comparing the results. The greatest difficulty in this process is determining whether two entries are comparable.

If this could be done independently of the file and directory structure, it would be relatively simple. Matching entities generally can be identified on the basis of class membership, name and signature. Static declarations, however, make trying to match file and directory names unavoidable. Moreover, it seems that organizing the final report in terms of the changes detected in particular files makes the reports easier to read and easier to use. For this reason alone, Cdiff must be able to correlate file names and directories used in the old program version with those in the new version.

Unfortunately, information about environment changes over the life of a program is not encoded in the program itself, so there is no way to extract the correspondences when the databases are built. This information has to come from outside. Happily, there are two simple heuristics that seem to cover many cases. The first of these is to do strict matching. Strict matching assumes that the directory structure has not changed between versions of the system, so files are correlated if they have the same name and the same directory path. This approach works well for programs that are evolving "in place". Cdiff uses this method of correlation as a default.

The second heuristic assumes that the directory structure has changed, but that the files have unique names. In this case files correlate on the basis of the file name alone. The Cdiff option `-basename` invokes this kind of matching. This can work well, even if some library headers violate the name uniqueness rules, so long as the user either filters out reports on library changes or restricts

the scope of the queries. The `-b` option is provided mainly as a way of doing quick experimentation. The output using this option reports on any changes in directory structure that Cdiff suspects.

There are cases in which neither of these two approaches produce good results. Any program that exhibits drastic directory structure changes or contains multiple source files sharing one name is too complex for the simple heuristics. In this case, Cdiff needs external help. The user must specify a list of path equivalences from the old program to the new program. This can be done either using the `CIAPATHEQ` environment variable or the `-patheq` option. The appendix shows the path equivalence initialization for InterViews.

All of these approaches deal with the problem of changing directory structure. However, sometimes the file names change as well. Currently, Cdiff will not attempt to correlate any files with different names. This generates spurious add and delete reports. Ultimately, Cdiff will need to allow a specification of known file name changes.

### 5.3 Evaluating Performance

Making any meaningful comparisons of the performance of Cdiff as opposed to textual differencers, like diff, is nearly impossible. The context of these tools is radically different and the kind of information they extract is barely comparable. I work on a SparcStation 1, running SunOs 4.1<sup>7</sup> that relies on a fairly congested network for file access. On this system, the Cdiff command:

```
Cdiff - -
```

that extracts all the symbol differences for all of the files in the databases takes about six minutes of real time to run. It takes one minute of real time to run diff on all of those files, but the time measured to execute diff does not reflect the 45 minutes it took to find all of the relevant file pairs and to write the shell script that would execute diff on those pairs. If I had not had the support of CIA++, it would have taken even longer to identify the relevant files. Then, once the diff script is finished, it would have taken hours of manual work to extract information about symbolic differences from the 7000 lines of textual differences generated by diff.

Normal use of Cdiff would involve queries with a smaller compass than the broad query above. The command:

```
Cdiff - ivTile:.* df=df ms=pb
```

that was demonstrated above takes about 30 seconds on the same network. There is no way to use diff to automatically generate the same information that the Cdiff command generates.

## 6 Conclusion and Status

Cdiff allows a programmer to quickly identify the significant parts of a program that have changed in terms of recognizable design and implementation components. This makes it possible to rapidly understand the nature of the changes without resorting to laborious cross checking between files of textual differences and versions of the code text.

Cdiff is the newest component of the CIA++ toolkit. It exists as a working prototype that will be refined through continued testing and use. It will also be used as a foundation for generating additional tools to investigate program evolution. This has been the typical evolution of all the CIA++ tools.

Additional tools that should be developed are tools that allow side by side comparison of the old and new versions of a modified symbol definition. There are existing CIA++-based browser tools that can be used to support a comparison browser.

There are a number of tools in the CIA++ toolkit that generate graphical views of structural relationships (for example: call graphs, inheritance graphs). These can be combined with Cdiff to

<sup>7</sup>Sparc and SunOS are trademarks of Sun Microsystems, Inc.

illustrate the impact of changes on the overall structure of a program and to track additional symbols that are indirectly affected by modifications.

The CIA++ system and its toolkit are currently being used within AT&T. Cdiff is new and not yet being widely distributed. The CIA++ system is available for universities under a binary code license agreement for academic institutions. The CIA++ package includes the CIA++ toolkit. Cdiff will be added to that package soon.

## References

- [1] Y. F. Chen. The C Program Database and Its Applications. In *Usenix Summer 1989 Conference Proceedings*, Baltimore, MD, 1989.
- [2] Y. F. Chen, M. Nishimoto, and C.V. Ramamoorthy. The C Information Abstraction system. *Transactions on Software Engineering*, 16(3):325-334, March 1990.
- [3] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, MA, 1990. ISBN 0-201-51459-1.
- [4] J. E. Grass. Object Oriented Design Archaeology with CIA++. *USENIX Computing Systems Journal*, 5(1), 1992.
- [5] J. E. Grass and Y. F. Chen. The C++ Information Abtractor. In *Usenix C++ Conference Proceedings*, pages 265-278, San Francisco, CA, April 1990.
- [6] M. A. Linton and P. R. Calder. The Design and Implementation of InterViews. In *Usenix C++ Workshop Proceedings*, pages 256-267, Santa Fe, NM, November 1987.
- [7] M. A. Linton, J. M. Vlassides, and P. R. Calder. Applying Object-Oriented Design to Structured Graphics. In *Usenix C++ Conference Proceedings*, pages 81-94, Denver, CO, October 1988.
- [8] M. A. Linton, J. M. Vlassides, and P. R. Calder. Composing User Interfaces with InterViews. *Computer*, 22(2):8-22, February 1989.
- [9] W. F. Tichy. Design, Implementation, and Evaluation of a Revision Control System. In *Proceedings of the 6th International Conference on Software Engineering*, Tokyo, Sept. 1982. IEEE.

## Appendix A An Initialization Shell Script for InterViews

This is a Korn Shell script to set the environment for differencing queries about InterViews. The CIAPATHEQ variable is a string giving pairs of directories that are to be considered equivalent. Each equivalent pair is written in the format: `old_directory:new_directory`. Semicolons are used to separate pair entries. Files with the same name in equivalent directories are considered to be comparable.

```
export OLDCIADIR=/home/grass/IVDB/IV3_B # database for 3.0 beta
export NEWCIADIR=/home/grass/IVDB/IV3_0 # database for 3.0.1

# Directory equivalencies... truly hideous!

export CIAPATHEQ="/n/gryphon/g2/InterViews/iv/src/include/Dispatch:\
/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/include/Dispatch;\
/n/gryphon/g2/InterViews/iv/src/include/InterViews:\
/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/include/InterViews;\
/n/gryphon/g2/InterViews/iv/src/include/OS:\
```



```
/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/include/OS;\n/n/gryphon/g2/InterViews/iv/src/include/InterViews/2.6:\n/src/X11R5/contrib/toolkits/InterViews/iv/src/include/InterViews/2.6;\n/n/gryphon/g2/InterViews/iv/src/lib/InterViews:\n/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/lib/InterViews;\n/n/gryphon/g2/InterViews/iv/src/include/InterViews/2.6/InterViews:\n/usr/local/src/X11R5/contrib/toolkits/InterViews/iv/src/include/InterViews/2.6/InterViews"
```



# C++ in a Changing Environment

Andrew J. Palay

*Silicon Graphics Computer Systems*

*ajp@sgi.com*

## Abstract

Current C++ systems have been designed without considering the requirements of environments that make use of shared libraries or dynamic loading. In these environments it must be possible to release new compatible versions of libraries or dynamically loaded components without recompiling portions of the system that make use of the classes defined in these new components. This paper describes our initial work on developing a new C++ system, called  $\Delta$ C++, that supports class changes with minimal recompilation. With  $\Delta$ C++, applications linked against a shared library will continue to run, without recompilation, even when a new version of the shared library is released.  $\Delta$ C++ can also be used to reduce the edit/compile/debug development cycle and provides a clean way to separate interface specifications from implementation.

## Introduction

The computer industry is moving toward developing software environments that make heavy use of shared libraries and dynamic loading. Shared libraries allow applications to reuse large amounts of code, thus reducing both memory and disk usage. Shared libraries can also be used by library providers as a mechanism by which they can release new compatible versions of their libraries without requiring applications to be rereleased. Existing applications will automatically see the benefit of using the new library. Dynamic loading can be used to build extensible software systems where developers are encouraged to build relatively small components that can easily be incorporated into already existing applications. This technique has been used in both the Andrew Toolkit [1] and NextStep [2] to support extensible multi-media user interface systems. For systems like this to be successful it must be possible for a user to install and use a component created after the original application into which it is going to be linked.

Releasing new compatible versions of code is one of the keys to supporting both shared libraries and dynamic loading. In the case of shared libraries it must be possible to release a new version of a library without invalidating the applications that use that library. In the case of dynamic loading it must be possible to release a new application without invalidating the loadable components. Similarly, it must be possible to release a new version of a loadable component without invalidating other components that use it. This is not the case with code generated by current C++ systems. A developer must be able release a new library or component that includes compatible changes to a class interfaces. Unfortunately current C++ systems require the recompilation of all code that depend on those changed classes. This results from the fact that current C++ systems resolve all object references at compile time, while in the above environments the information needed to do that resolution is not known until link time..

Over the past year we have been working on a new C++ system, called  $\Delta$ C++, that solves this problem. We developed a prototype version of  $\Delta$ C++ that has been used to understand the problems that arise when trying to support C++ in a changing environment. In particular, we were interested in understanding the set of object references that must be resolved at link time and how

they should be resolved. We were also interested in understanding whether we can support this level of dynamics without changes to the C++ language (the answer is yes). This paper describes the initial results of this work. We discuss the types of class changes we wish to support and the specific problems that must be solved by C++ systems in order to support these changes. We also describe the runtime solution used by our prototype version of  $\Delta$ C++, followed by a brief discussion of future work we plan to do toward developing a linktime version of  $\Delta$ C++ that solves the same problems but without the performance penalty incurred by the prototype. Finally we discuss some other problems that can be solved using the  $\Delta$ C++ technology.

## Types of Compatible Class Changes

### *Changes that must be supported*

In discussing the types of changes we begin with the following definitions for the classes Alpha and Beta:

```
class Alpha {
    public:
        long a;
        long A();
        virtual long B();
};

class Beta {
    public:
        long x;
        virtual long X();
};
```

There are four major types of changes to class interfaces that we believe must be supported. The simplest change is *member-extension*. It must be possible to add both member functions and variables to a class without forcing the recompilation of any code that uses that class. This must be true for public, protected and private members. For example we must be able to release a new version of Alpha that has the following interface without requiring code that uses Alpha to be recompiled:

```
class Alpha {
    public:
        long a;
        long b;
        long A();
        virtual long B();
        virtual long C();
};
```

A second form of extension that must be supported is *class-extension*. It must be possible to add a new base class to an already existing class. In our example, we must be able to add Beta as a base class to Alpha. While Alpha will now support additional functionality, it still supports its original interface.

Another type of modification that needs to be supported is *member-promotion*. This is the moving of functionality from a derived class to a base class. Given that Alpha is derived from Beta, a developer must be free move some of the members from Alpha into Beta. The new version of Alpha will still provide a compatible interface. Users of Alpha should not be interested in how the functionality of Alpha is provided only that it is provided. Thus a developer should be able to release the following new versions of Alpha and Beta without requiring any code that uses either class to be recompiled:

```

class Alpha : public Beta {
    public:
        long a;
        long A();
        virtual long B();
};

class Beta {
    public:
        long x;
        long b;
        virtual long X();
        virtual long C();
};

```

The last major form of modification that we need to support is *override-changing*, especially for member functions. A user of Alpha should be unconcerned whether Alpha overrides the member function X(), originally declared in Beta. The function that will get called when invoking X() on an instance of Alpha will change but the code should still work. A similar case can be made for member variables, although we think supporting *override-changing* of member variables needs to be examined in greater detail.

Given we need to support the above types of modifications, two other modifications, *member-reordering* and *class-reordering*, can be easily be supported. The location of a member in a class need not be fixed for all time. A developer might wish to reorder the member variables in order to make more efficient use of space, or may choose to reorder members in order to group members by their protection level. This reordering has no effect on the interface being provided by the class.

### *Changes that need not be supported*

There are a number of modifications that we have chosen not to support. In each case, supporting such a change would require code modifications that can not be done in an efficient manner. These modifications include:

- changing the inheritance of a class from non-virtual to virtual or from virtual to non-virtual.
- changing a method from non-virtual to virtual or from virtual to non-virtual.
- widening the type of a member (eg. from short to long).

### *Changes that we believe should be supported*

There are a set of modifications that we can possibly handle, although there is some disagreement as to whether we should. In each of these cases we currently believe that the modification should be supported but we also understand that a case can be made for not supporting it.

Changes to inline functions are the first such modification which can not be handled without modifying the code that is generated. The only effective method for handling changes to inline member functions is not to inline the functions in the first place. This will have a detrimental effect on the performance of some programs, thus the disagreement on whether it is right to support inline modifications. In reality, we probably can not completely eliminate the use of inline functions, so we will need to educate developers when it is appropriate to use them.

Changing the value of a default parameter to a function is a second controversial type of modification. The controversy is not over performance considerations but with the programmer's understanding of default parameters. A developer can either view the default parameter specification as a statement that the compiler will provide a default (which just happens, for implementation reasons, to be listed in the class specification) or that the compiler will provide the specific default. In the former case the default specification is just a shorthand for declaring and defining a several methods. In the latter case the developer uses the default specification as a shortcut when entering a program. The developer could have typed in the full call with all the

parameters specified, but as long as the defaults provide the proper values, those parameters can be skipped. The C++ language, with its syntax for specifying default parameters, makes it impossible to determine which of these interpretations is correct.

Changing the assignment of constant values within a class is another controversial type of modification. This problem arises when an enumeration with assigned values is declared within a class. The controversy is similar to the problem with default parameters. There is no way to decide whether the values of those constants are part of the visible interface.

### *Is handling all these modifications necessary?*

It can be argued that we are being too adventurous relative to the types of modifications that we wish to support. When people initially look at this problem they focus on the problem of supporting changes to just the private portion of an class interface. This is quite understandable, since the private portion should have no effect on users of a class. A typical solution to this problem is to move what was the private portion of a class into a separate implementation class, and have the original class's private portion contain just a pointer to an instance of that implementation class. While this solves the problem of handling changes to the private portion of the class, it does not solve the problem of adding functionality to a class, nor changing the overriding behavior of the class. When dealing with releasing new versions of shared libraries and dynamic loadable components, functionality improvements are at least as important as handling changes to the private implementation of a class. Arguing this is similar to arguing that new releases of an operating system should not add any functionality, but only include performance improvements that can be made without effecting the public interface provided by the operating system.

### **Problems to be Addressed**

As stated earlier, the basic problem with current C++ systems, is that object resolution is done at compile time. In order to support the types of modifications presented in the previous section the earliest object resolution can be done is link time. The first problem we must solve is determining the exact layout of a class instance. Object code must contain enough information about the class definitions to do that layout. This includes the set of base classes and the member variables to be added by the class. It must be possible to determine the size and required alignment of each member variable. For basic types the size can be provided by the compiler, while for instances of other classes that size must be resolved dynamically. The process of determining the layout of a class will determine the following:

- the size of a class instance
- the offset for each class member
- the location (if any) of any vtable pointers
- the location (if any) of any virtual base class pointers.
- offsets to move between classes.

The offset for each class member will be used to resolve any reference to a member variable. For example, returning to the example presented in the previous section, the expression `pa->a`, where `pa` is a pointer to an instance of `Alpha`, will need to access data at a different offset from `pa` depending on which definition of `Alpha` is being used. The vtable pointers are needed for calling virtual methods and for the code added to constructors and destructors. The virtual base class pointers are needed to access data in any virtual base classes and also for code in the constructors. The interclass offsets are needed to support casts and are also used to determine the offsets associated with member functions.

Just as we need to be able to determine the layout of a class instance, we also must be able to determine the layout and values stored in the vtable for a class. Again, this requires that the object code contains the list of base classes and the member functions provided by the class. From this information, the following will be done:

- allocation of the class's vtable
- determination of offsets for each member function
- initialization of function and offset fields for each vtable entry.

The offset for a member function is used to determine the proper slot in the vtable to be used when calling a virtual member function. The initialization of the vtable requires that we are able to determine the overriding behavior of the classes involved and, in the case of multiple inheritance, be able to set the offset field for each function entry to adjust the pointer to the instance appropriately.

In the process of doing object resolution we must also be able to determine the actual functions that will be called when invoking either a non-virtual member function or a static member function. In the case of a non-virtual member function and multiple inheritance we must also determine the offset associated with that function. In both of these cases we need to adjust any calls to those functions to invoke the correct function. In a similar fashion we need to resolve references to static member variables.

Proper function resolution must also be done for potentially generated functions like constructors, destructors, and operator =, as well as for operator new and operator delete. Further the code added to constructors and destructors must be ready to handle an arbitrary set of base classes (both non-virtual and virtual) since we will not know *a priori* a class's set of base classes.

Another problem that must be solved is the allocation of global, static or automatic class instances. Since the size of the class is not known until at least link time the allocation of those instances must be delayed. For global and static instance this is not a major problem, whereas automatic instance are. Changes in the size of an automatic instance will potentially change the offset for other, perhaps non-class, variables that are to be allocated on the stack.

## Runtime ΔC++

The prototype runtime version of ΔC++ provides a solution to the above problems, but with a substantial performance penalty (about a factor of 2 in both size and speed). We built this prototype in an attempt to understand the problems that must be addressed and also to understand how we should build a link time solution to these problems. It is not intended to be used as a real solution to the problems being addressed in this paper. By describing some of the code translations from C++ to C that we used, we hope to provide a better understanding of the problems that must be addressed when adding dynamics to the C++ environment..

Runtime ΔC++ utilizes a set of *offset* variables (one per member) to resolve references to class members. These variables are initialized as part of the process that resolves class definitions which takes place as part of the initialization phase before a C++ application really executes. In the following example:

```

class Alpha {
public:
    long a;
    long A();
    virtual long B();
};

Alpha *pa;

```

the code generated for the expression `pa->a` looks like:

```
*((long *) ((char *) pa) + __mtable_Alpha[__a_Alpha.vo])).
```

The vector `__mtable_Alpha` contains the offsets that are needed to access members of a class and the variable `__a_Alpha.vo` indicates the slot in that table that will hold the offset value for the member `a`. In this case `__mtable_Alpha[__a_Alpha.vo]` is initialized to be the number of bytes that the member `a` is away from the start of the class `Alpha(0)`. It may seem that the use of `__mtable_Alpha` is unnecessary, however it is needed if we wish to allow for the overriding of member data. For example, if the class `Gamma` is derived from `Alpha` and we wish to allow `Gamma` to be redefined at a later time to also have a member `long a`, and have references of the form `pg->a` where `pg` is a pointer to a `Gamma` now refer to the `a` in `Gamma` instead of in `Alpha`, then using the vector `__mtable_Alpha` is necessary. If we do not want to allow overriding of member data then the expression `pa->a` would result in the following code:

```
*((long *) ((char *) pa) + __a_Alpha.offset))
```

where `__a_Alpha.offset` would contain the number of bytes that the member `a` is away from the start of the class `Alpha`.

The expression `pa->A()` generates:

```

(*(long *) (struct Alpha *)) (__vtable_Alpha[__A_Alpha.mo].f))
((struct Alpha *) ((char *) pa)
+ __A_Alpha.so
+ __vtable_Alpha[__A_Alpha.mo].d)))

```

and the expression `pa->B()` generates:

```

(*(long *) (struct Alpha *)) (((struct __mptr **)
((char *) pa) + __vptr_Alpha)[__B_Alpha.mo].f)
((struct Alpha *) ((char *) pa) + __B_Alpha.so
+ *((struct __mptr **) ((char *) pa)
+ __vptr_Alpha))[__B_Alpha.mo].d))).

```

The scheme for calling virtual and non-virtual member functions is quite similar. In each case we are choosing to do the function lookup through a method table. In the non-virtual case we are doing a lookup through the method table that is associated with the declared class of the calling object. In the virtual member function case we are using the method table attached to the calling object. In each case the function that is going to be called is determined by a variable, `__A_Alpha.mo` and `__B_Alpha.mo`, respectively. These variables give the offset into the method table that holds the appropriate function and the value to add to the variable `pa` to handle the case of multiple inheritance. The variables `__A_Alpha.so` and `__B_Alpha.so`, which would be set to 0 in this example, are needed to handle changes in the class structure that introduce multiple base classes. The variable `__vptr_Alpha` gives the offset (in bytes) to the `vptr` for the class `Alpha`.



The above code may seem to be overly complex but each field is necessary in order to handle the modifications described above. For example the above code must continue to work if the definition of Alpha is changed to:

```
class Alpha {
public:
    long al;
    virtual long C();
    long a;
    long A();
    virtual long B();
};
```

In this case the value assigned to `__a__Alpha.vo` would change from 0 to 1, the value of `__mtable[__a__Alpha.vo]` would change from 0 to 4, the variable `__A__Alpha.mo` would change from 0 to 1, and the variable `__B__Alpha.mo` would change from 1 to 2. With these changes the above code fragments will continue to function correctly.

These fragments will continue to work even if we choose to further modify our example so that the definition of Alpha looks like:

```
class Aleph {
public:
    long al;
    virtual long C();
};

class Beth {
public:
    long bt;
    long a;
    long A();
    virtual long D();
    virtual long B();
};

class Alpha : public Aleph , public Beth {
public:
    long B();
};
```

In this case the value of `__a__Alpha.vo` would change to 3 and the value `__mtable_Alpha[__a__Alpha.vo]` would change to 12 (the 2nd slot of `__mtable_Alpha` would hold the location of the vptr which is located 4 bytes into the instance). The value of `__A__Alpha.so` would change to 8, and the function stored in `__vtable_Alpha[__A__Alpha.mo].f` will be changed from `Alpha::A` to `Beth::A`. The change to the value `__A__Alpha.so` makes sure that the function `Beth::A` will be called with the correct value for `this`. The value of the variables `__B__Alpha.mo` and `__B__Alpha.so` will be changed to 3 and 8 and the value of `__vtable_Alpha[__B__Alpha.mo].d` will be changed to -8. Thus, if `pa` is really of type `Alpha` then the code `pa->B()` will invoke the function `Alpha::B` with `pa` passed as the first parameter,

So far we have shown how we can handle, for non-static members, *member-extension*, *class-extension*, and *member-promotion*. *Member-reordering* and *class-reordering* are simple, since the actual determination of a member's location is always done via a variable. Changing the position of a member is handled by changing the value of the appropriate offset variables and tables. The same is true for changing the order of the base classes.

The above code fragments also handle *override-changing*. If we change the last definition of Alpha so that it has the following definition:

```
class Alpha : public Aleph , public Beth {
public:
    long a;
    long A();
    long B();
};
```

The value of `__a_Alpha.vo` remains the same, however the value of `__mtable_Alpha[__a_Alpha.vo]` will change to 20. Similarly the values of `__vtable_Alpha[__A_Alpha.mo].f` and `__vtable_Alpha[A_Alpha.mo].d` will change to `Alpha::A` and `-8`, respectively.

In the previous examples we have examined the code that must be generated for non-static members of a class. It must also be possible to handle the same types of modifications to the static portion of a class definition. If we extend the class Alpha to contain two static members, `static long s` and `static long S()`, then the code generated for `pa->s` is:

```
((long *) __stable_Alpha[__sa_Alpha.vo]))
```

where `__stable_Alpha` is a vector that contains pointers to the static variables for the class Alpha. Thus, in this example `__s_Alpha.vo` would be 0 and `__stable_Alpha[0]` contains `&Alpha::s`. Similarly the expression `pa->S()` generates the following code:

```
(((((long *) (void)) (__vtable_Alpha[__S_Alpha.mo]).f))))()
```

The use of the vectors, `__stable_Alpha` and `__vtable_Alpha`, and the offset variables, `__s_Alpha.vo` and `__S_Alpha.mo`, provide the level of indirection that is necessary to handle the required modifications. If we promote the member `s` into the class Aleph, the value of the variable `__s_Alpha.vo` may change (depending on whether we have added other static member variables), but the value of `__stable_Alpha[__sa_Alpha.vo]` will change to `&Aleph::s`.

The above examples give a slightly misleading picture of the code that is generated for the above class definitions. The code fragments listed above would only have been generated if the code was compiled using the original definition of Alpha. If we recompile the same code after we added the base classes of Aleph and Beth, the original code fragments `pa->a`, `pa->A()`, and `pa->B()` change to the following:

```
((long *) (((char *) pa)
+ __mtable_Alpha[__Beth_Alpha.vo + __a_Beth.vo]))

((long *) (struct Alpha *))
(__vtable_Alpha[__Beth_Alpha.mo + __A_Beth.mo].f))
(((struct Alpha *) ((char *) pa)
+ __Beth_Alpha.offset
+ __A_Beth.so
+ __vtable_Alpha[__Beth_Alpha.mo + __A_Beth.mo].d)))
```

```

(*(long (*)(struct Alpha *)) (*((struct __mptr **)
((char *) pa) + __vptr__Alpha))(__Beth__Alpha.mo + __B__Beth.mo).f)
((struct Alpha *) ((char *) pa)
+ __Beth__Alpha.offset + __B__Beth.so
+ (*(struct __mptr **) ((char *) pa)
+ __vptr__Alpha))(__Beth__Alpha.mo + __B__Beth.mo).d))).

```

where the values of the variables are as follows:

```

__Beth__Alpha.vo = 1
__a__Beth.vo = 1
__mtable__Alpha[2] = 12
__Beth__Alpha.mo = 1
__A__Beth.mo = 0
__vtable__Alpha[1].f = Beth::A
__Beth__Alpha.offset = 8
__vtable__Alpha[1].d = 0
__B__Beth.mo = 2
__vptr__Alpha = 4
__vtable__Alpha[3].f = Alpha::B
__vtable__Alpha[3].d = -8.

```

In this example we have introduced another structure `__Beth__Alpha` which contains information that is used to move up the class hierarchy from Alpha to Beth. The `offset` field provides the number of bytes that must be added to an instance of an Alpha to change it into a Beta. The `vo` and `mo` fields are used to allow extensions to a base class to be made without recompiling any derived class. We want to be able to add a new member to the class Beth and, without recompiling the code for Alpha, and be able to reference that new member from an instance of an Alpha. When we add a member to a class, we require that the code that provides the offset variables for that class be recompiled. This guarantees that the appropriate offset variables will be defined. This is not true for derived classes, which need not be recompiled when a base class changes. For example, if we add the member `long Bt()` to the class Beth then the structure `__Bt__Beth` will exist and be properly initialized, whereas, until Alpha is recompiled, the structure `__Bt__Alpha` won't exist. The `vo` and `so` fields of the `__Beta__Alpha` structure make it possible to access the proper parts of the `__mtable__Alpha` and `__vtable__Alpha` tables.

The need to handle this kind of change also explains why we must reference all members through a function table. If we could assume that the structure `__Bt__Alpha` exists whenever Alpha contains the member `long Bt()`, even if it is inherited from a base class, then a call to a non-virtual member function could look like:

```

(*(long (*)(struct Alpha *)) (__A__Alpha.mo.f))
((struct Alpha *) ((char *) pa)
+ __A__Alpha.so
+ __A__Alpha.d)))

```

Unfortunately the above assumption can not be made, thus dictating the use of the function table.

C++ allows a programmer the ability to directly reference a member of a class by providing its qualified name. Thus we need to be able to handle expressions of the form `pa->Beth::a` and `pa->Alpha::C()`. Each of these cases result in code that first casts the pointer `pa` to the qualified class and then uses the code given above for instance variables and non-virtual member functions. The generated code for casting an Alpha to a Beth looks like:

```
((struct Beth *) (((char *) pa) + __Beth__Alpha.offset))
```

and the code generated for the expressions `pa->Beth::a` and `looks pa->Aleph::C()` looks like:

```
*((long *) (((char *) pa) + __Beth__Alpha.offset
+ __mtable_Beth[__a__Beth.vo]))

(*( (long *) (struct Beth *)) (__vtable_Beth[__C__Beth.mo].f))
(((struct Alpha *) (((char *) pa)
+ __Beth__Alpha.offset
+ __C__Beth.so
+ __vtable_Beth[__C__Beth.mo].d))).
```

Virtual base classes further complicate this picture in that it introduces another level of indirection. The major problem with virtual base classes arises when a derived class overrides either a non-virtual member function or a member variable. Since the location of the virtual base class instance relative to the top of the class instance can only be computed dynamically, there is no easy way to get the `this` to point to the proper place. If we extend our example so that both `Aleph` and `Beth` derive virtually from the class `Gimmel`, which contains an non-virtual member function `long G()` then calling the function:

```
long test(Aleph *pal)
{
    return pa->G();
}
```

should result in calling the function `Gimmel::G` with a pointer to a `Gimmel`. If in a later release `Aleph` overrides `G()` then the call will have to result in a call to `Aleph::G` with a pointer to an `Aleph`. The code generated for `test` must handle both of these cases. The way the runtime version of  $\Delta C++$  handles this problem is by generating code that casts `pal` to a `Gimmel` and then calling the appropriate method. Unfortunately there is not a fixed offset that should be added to the casted version of `pa` in order to convert it back into an `Aleph`. The offset that is used when passing in an `Aleph` is different than the offset used when passing in an `Alpha`. Determining the appropriate offset, requires that we select the offset using the type of the object passed in as a parameter. The type of an instance is stored in the first entry in `vtable` attached to the object (this does require that every instance have a `vtable` pointer).

## Future Work

As we stated earlier, the runtime version of  $\Delta C++$  is not an adequate solution to the problems of supporting shared libraries and dynamic loading in  $C++$  as the performance overhead is too great. No developer will accept a factor of 2 decrease in performance in order to solve these problems. We are developing a compile/linktime solution to this problem that provides the above functionality without any major performance penalty. The basic concept is to replace the multitude of variables used in the runtime solution with a set of relocation types in a linktime solution. Efficient code for member access will be generated, and modified at link time with the proper offsets. The same work with respect to the generation and initialization of `vtables` will also be done by the linker.

We have also started to look at how we support templates in this same environment. The mechanisms outlined in this paper solve one of the problems associated with templates. Using  $\Delta C++$ , a developer will be able to change the template definition in exactly the same ways as normal classes. The problem with templates that still needs to be addressed is how and when

should the template be instantiated. Using the current *cfront* 3.0 solution, where instantiation is done when the application is linked together, is not viable. With shared libraries, linking is done when the user runs an application. Since the user may not have access to the compiler nor would the user want to wait for the compiler to run, generating code at that time is not feasible.

## Other Uses of $\Delta$ C++ Technology

We started working on  $\Delta$ C++ in order to solve the problems of shared libraries and dynamic loading. During the development of the prototype we started to realize that there are other places where this technology can be used. We have looked at using this technology to solve several problems in the application development process. Using  $\Delta$ C++, developers need not recompile large portions of their application whenever they change (or someone else changes) a class definition. When a class definition is changed, only the code that implements that class, that uses the new parts of the definition, or that used parts of the definition that has been removed need to be recompiled. This should greatly reduce the edit/compile/debug cycle. In a similar fashion  $\Delta$ C++ will allow base class developers the ability to modify class definitions and test their changes against already existing code. With current C++ systems, the cost of changing a class definition high up in the hierarchy is often prohibitive. In order to verify that the change does not cause any problems all the code that uses that base class must be recompiled. In developing the Andrew Toolkit, which had a similar compile time model as C++, we discovered that developers eventually refused to make changes in important base classes, since the cost of testing was so high. The result of this that developers would work around problems in the most basic parts of the system that should have been fixed. With  $\Delta$ C++ the developer can change the class definition, recompile a few files and relink the resulting applications to test the change. This fundamentally changes the cost of making basic changes in the system.

Another use of  $\Delta$ C++ is delayed binding of class implementations.  $\Delta$ C++ allows us to completely separate interface and implementation thus solving the problem discussed in [3] without creating lots of additional classes nor using multiple inheritance. Using either shared libraries or dynamic loading a user can choose which implementation of a class should be used when running an application. The choice can include using implementations that were not released with the application but provided later by another developer. For example, a user could select the presentation style to be used by an application by pointing the application at different libraries, each which supports the proper interfaces but with different class interfaces and implementations. In this way an application could be shipped with a Motif look and feel and someone else could provide an OpenLook look and feel at a later time. Further subclassing with the application code from those components would still work.

Using  $\Delta$ C++ also allows library providers the option of eliminating the private portion of a class interface when releasing their product. The code compiled with an abridged version of a class definition will always work with an implementation compiled with a complete version.

## Related Work

The issue of dynamic loading in C++ has been addressed in previous papers [4, 5], but ignored the issues raised in this paper. We believe that the combination of  $\Delta$ C++ and the ideas presented in these papers can lead to a usable dynamic loading system.

Environments such as Lisp and Smalltalk that support the problems described in this paper have existed for a long time. This work differs from these systems in that the ultimate goal is to develop an environment that supports the full C++ language specification [6] with little performance degradation over standard C++ systems.

## Conclusion

Current C++ systems have been designed with the assumption that the end-user of an application has access to both the sources of the application (and possibly its supporting libraries) and to a C++ compiler. Current systems also have been designed with the assumption that the cost of running the compiler is essentially zero. While it may be argued that these assumptions were valid in the past, they are no longer valid today. In the today's world, library and application developers have no desire to release source code for their product. Even if they did end-users probably have not purchased the compilation environment nor would they want to run the compiler if it was bundled on their system. We must develop software that can be purchased, installed and run with limited overhead. Our work with  $\Delta$ C++ is an initial attempt to look at these problems. We have attempted to solve what we believe to be the biggest problem, the changing of class definitions with limited recompilation. We already know that template instantiation is another problem area we need to investigate. In the future any proposed extension to C++ must consider the problems raised by environments that make use of shared libraries and dynamic loading.

## Acknowledgments

I would like to thank David (Bean) Anderson and Mark Linton for allowing me to bend their ears about this work. This was especially helpful when trying to make sense of the complexities of C++ and cfront. I would also like to thank Miles Bader (a former colleague of mine when I worked at the Information Technology Center, Carnegie Mellon University), who did some very early work at solving similar problems in the object system used in the Andrew Toolkit.

## References

- [1] Mark Sherman, David Anderson, Wilfred J. Hansen, Thomas P. Neuendorffer, Andrew J. Palay, and Zalman Stern, "Allocation of User-Interface Resources in the Andrew Toolkit". In *Proceedings of the International Conference on Multimedia Information Systems '91*, McGraw Hill, 1991, pages 261-272.
- [2] NeXT System Reference Manual, NeXT Inc.
- [3] Bruce Martin, "The Separation of Interface and Implementation in C++". In *Proceedings of the 1991 USENIX C++ Conference*, 1991, pages 51-62.
- [4] Sean M. Dorwood, Ravi Sethi, and Jonathan Shupiro, "Adding New Code to a Running C++ Program". In *Proceedings of the 1990 USENIX C++ Conference*, 1990, pages 279-292.
- [5] B. Stroustrup, "Possible Directions for C++", *1987 USENIX C++ Workshop*, 1987, pages 399-416
- [6] M Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

# Adding Concurrency to a Programming Language

Peter A. Buhr and Glen Ditchfield

*Dept. of Computer Science, University of Waterloo,  
Waterloo, Ontario, Canada, N2L 3G1  
{pabuhr,gjditchf}@plg.uwaterloo.ca*

## Abstract

A programming language that lacks facilities for concurrent programming can gain those facilities in two ways: the language can be extended with additional constructs, which will reflect a particular model of concurrency, or libraries of types and routines can be written with different libraries implementing different models. This paper examines the two approaches, for object-oriented and non-object-oriented languages. Examples show that concurrency interacts extensively with traditional programming language constructs, and that general elementary facilities for concurrency must be implemented at extremely low levels—the assembly language level, in some cases—and hence that safe support for concurrency requires language extension.

## 1 Introduction

To take advantage of asynchronous hardware, such as I/O devices or multiple CPUs, a programming language must provide the ability to interact with other programs without blocking, such as calls to operating system I/O routines, or to start multiple independent operations so that if some operations block others can continue to make progress. Traditionally, programming language concurrency has been available only by interacting with the operating system, usually with each task in a different address space. In general, this organizational structure makes creation of new processes and communication among tasks expensive [ABLL92]. Users will not make the additional effort to design and write concurrent programs if the complexity required is too great or the performance pay back is too small. Therefore, new programming languages must provide concurrency and existing programming languages must be augmented with concurrency if they are to be useful in a parallel environment. Finally, concepts that lead up to concurrency, such as coroutines, allow certain kinds of problems, such as finite state machines and push-down automata, to be expressed in eloquent and straightforward ways.

Can concurrency be provided by library definitions built from existing language constructs? If not, what language constructs are needed to make this possible? Would those constructs be useful for purposes other than concurrency? This paper examines programming language facilities that must exist to implement concurrency in object-oriented and in non-object-oriented programming languages. The purpose is to determine if the fundamental aspects of concurrency can be provided through generally available language constructs, or if concurrency requires languages to be augmented with additional constructs.

Much of this analysis comes from our work in adding concurrency to C++, which resulted in a new dialect called  $\mu$ C++ [BS92] that extends C++ with several new language features.  $\mu$ C++ has been criticized for extending C++ instead of adding concurrency using existing language features. This paper attempts to deal with this criticism by showing that it is *not* possible to build concurrency facilities from existing language features in C++ without sacrificing essential features. As well, this discussion should be useful to designers of new languages and those extending existing programming languages with concurrency features. Many ideas presented in this discussion appear in [BDS<sup>+</sup>92], but this paper presents a more general and thorough analysis.

A general knowledge of concurrency is assumed throughout this discussion.



## 1.1 Simplicity versus Complexity

We, like others, believe in small languages with strong abstraction facilities.

In particular, the language should get as much mileage as possible out of its definitional mechanism, never introducing something as a distinct language construct which can better be explained in terms of the definitional mechanism. [Hil83, p. 13]

For example, in C++ [ES90], dynamic memory allocation is provided by library operators `new` and `delete`. The default storage management facilities can be replaced by libraries that provide tracing or debugging features [ZH88, Cah], or garbage-collecting allocators [BW88], or allocators tuned to specific allocation patterns. In Pascal [JW85], storage management is provided by standard `new` and `dispose` routines. These operations are usually part of the compiler's run-time environment, so replacing them is difficult or impossible. In this respect, C++ is more flexible than Pascal.

Replacing primitives with programmer-definable facilities leads to a smaller, simpler language kernel, and simplicity is generally held to be a virtue in programming languages [Hoa73, Wir74]. However, a small kernel does not imply a reduction in the total complexity of the programming system. Therefore, the main advantage of the library approach is its flexibility. For example, different libraries can provide different models of concurrency, such as the Linda model [CG89] or the Actors model [Agh86]. However, even if a language is general enough to implement a variety of different concurrency models, it is doubtful that programs using different models can interact. Furthermore, issues of syntax and type safety must be dealt with when defining operations like `new` and `delete`. At best, a library will be as convenient and as safe as primitive language features. Therefore, as far as a user of a particular model is concerned, there is little difference between an extensible language supporting their model and a specialized language supporting their model.

## 2 Elementary Execution Properties of Concurrency

As discussed in [BDS<sup>+</sup>92], there are three elementary execution properties of concurrency:

1. A **thread** sequentially executes programming language statements, independently of and possibly concurrently with other threads. A thread's function is to perform a computation by changing execution-states.
2. An **execution-state** is the state information needed to permit concurrent execution. In practice, an execution-state consists of the data items created by an object, including its local data, local block and routine activations, and a current execution location. A programming language determines what constitutes an execution-state, and therefore, execution-state is an elementary property of the semantics of a language. (An execution-state is related to a continuation. Creating a continuation makes a copy of the current execution-state [HD90].) A **context switch** occurs when a thread switches from one execution-state to another.
3. **Mutual exclusion** is the mechanism that gives a thread sole access to a resource for a period of time.

The first two properties represent the minimum needed to perform execution, and seem to be fundamental in that they are not expressible in machine-independent or language-independent ways. For example, creating a new thread requires creation of system runtime control information, and manipulation of execution-states requires machine specific operations (modifying stack and frame pointers). Mutual exclusion is expressible in terms of simple language statements (for instance by implementing Dekker's algorithm), but doing so is error-prone and computationally expensive, and therefore we believe that mutual exclusion must be provided as an elementary execution property. Therefore, any programming language that supports concurrency must provide primitive constructs to implement these properties. While this can be done in a number of ways, additional design requirements of a programming language may impose additional constraints.



### 3 Design Options

Concurrency facilities must blend with other aspects of a programming language.

**form of a task** – Does a task resemble other language constructs? A non-object-oriented programming language might present a task as an independently executing program, analogous to the body of a non-concurrent program. An object-oriented programming language might present a task as an object, with an interface defined by a set of member functions.

**form of communication** – Does task communication resemble other language communication? Task communication might resemble a routine call, with data passed as arguments and received as parameters, or tasks might communicate through intermediate “channel” objects, which resembles file I/O. Alternately, communication could involve new operations to send and receive “message” objects, which programs must assemble and disassemble. Multiple forms of communication in a language can be confusing for users and inefficient because data must be transformed from one form to another along a communication path. We argue for using the routine-call mechanism in C++ because that is the form used to communicate with objects.

**static type-checking** – Can all communication in the language be statically type-checked? Languages with compile time (static) type-checking versus runtime (dynamic) type-checking have additional requirements on the communication mechanism. Sufficient definitions must be made and be available so that the compiler can type check all communication, especially across separate translation units.

**declaration scopes** – Are the concurrency features available or restricted by the declaration scopes of the language? For instances, if tasks resemble objects, then it should be possible to declare a task anywhere that an object can be declared.

A designer of concurrency facilities must choose between alternative ways of providing them.

**direct and indirect communication** – Can tasks communicate directly with one another or does all communication occur through a third party? If communication requires a third party, e.g. a monitor [MMS79, HC88] or tuple space [CG89], this can slow execution when a large number of tasks are interacting in a complex way because of additional synchronization and data transfers with the intermediate object.

**synchronization and mutual exclusion** – Is mutual exclusion and synchronization implicit and limited in textual scope, or explicit and tied to the flow of control? It is our experience that requiring users to build complex mutual exclusion facilities, like monitors, out of low-level mutual exclusion primitives, like locks, often leads to incorrect programs. Furthermore, we have noted that reducing the textual scope in which synchronization occurs reduces errors in concurrent programs.

**synchronous or asynchronous communication** – Both synchronous and asynchronous communication are needed in a concurrent system. In synchronous communication, a task that transmits information suspends execution until another task receives it and replies; in asynchronous communication, the transmitter may continue execution before the receiver picks up the data. Since synchronous communication can implement asynchronous and vice versa, a language need only provide one of the two mechanisms. We argue that a language should provide synchronous communication out of which asynchronous communication can be built. If asynchronous communication is the primitive mechanism, this usually implies the existence of variable-sized dynamically-allocated buffers. In general, this is too expensive a mechanism to be built into the language. Asynchronous mechanisms should be provided by library facilities like buffers and/or futures.

**order of processing requests** – An object that is accessed concurrently must have control over the order in which it services requests. Without this ability, all requests must be processed in first-in first-out (FIFO) order; any other order requires a programmer to devise a multi-step protocol. FIFO servicing may inhibit concurrency and has deadlock problems [Gen81], while protocols are error-prone because a user may not obey the protocol (e.g. never retrieve a result). The ability to postpone a request is sufficient, where postponing means that a task can accept a request, examine it, and decide not to perform it for an unspecified time, while continuing to accept new requests (available in Thoth [Che82], Harmony [Gen85], and the V-system [Che88]).

It is also extremely convenient and often more efficient if a concurrent object can also control which pending request it receives next (available in SR [AOC+88] and Concurrent C [GR89]), rather than having to receive requests in FIFO order and possibly postpone inappropriate ones. There are many situations where a concurrent object knows that it can service only a certain kind of request or a request from a certain object next. However, the ability to select a pending request is insufficient if servicing requires other resources that may not be immediately available; only the ability to postpone a request allows a task to continue servicing requests until the resources becomes available and the request can be completed.

We reject any solutions to these options that involve coding conventions or multi-step protocols because such solutions are error-prone both to implement and maintain.

## 4 Concurrency Libraries

The following sections examine the difficulties in adding concurrency through language definitions and routines assuming that some primitive mechanisms *already* exists to provide the three elementary execution properties. Both object-oriented and non-object-oriented programming languages are examined.

### 4.1 Starting a Library

Currently, C++ does not define the relative order of initialization of objects declared with static storage duration in different translation units. As a result, there is no way to ensure that a library is initialized before the objects that depend on it are instantiated (e.g. like the runtime system of a concurrency library). This leaves a library implementor with three options in C++:

1. Forbid the declaration of library objects with static storage duration.
2. Test repeatedly in the library to ensure that initialization is performed.
3. Declare an instance of a library initialization object before any declarations that depend on the library in each translation unit, but ensure that only one of the initialization instances actually performs the initialization of the library. The declaration of the initialization object can be made implicit by putting it in the include file for the library, which must be included in each translation unit before library features are used. (This is the approach used to start the  $\mu$ C++ runtime library.)

The first solution is very restrictive, the second is inefficient, and the third is a C++ idiom that is not obvious to a library implementor. This situation is handled properly by modules in other languages [Uni83, CDG<sup>+</sup>88], which define an order of initialization among modules.

### 4.2 Context Switching

C and C++ have a language facility called `setjmp/longjmp` that was introduced to provide a simple form of exceptional control flow. These routines save and restore execution-state to allow non-local gotos. This language feature has been used as the basis for context switching in some thread packages.

However, `setjmp/longjmp` can be inefficient for context switching. The problem occurs with co-processor data, such as floating point registers, and any other data that is task specific. Saving all of this data substantially increases the cost of context switches. For example, our test results show that on a Sequent Symmetry S27 (Intel 386) the context switch time can double when the floating-point registers are saved. Since many tasks do not use floating point, this can be of significant concern. Flow analysis in a compiler may determine that a task does not use the floating point registers so only the fixed point registers have to be saved on a context switch. Most light-weight tasking systems require the user to explicitly indicate whether the floating point registers should be saved on a context switch, which is error prone.

### 4.3 General Library Routines

In general, most UNIX library routines are *not* reentrant. For example, many random number generators maintain an internal state between successive calls, and there is no mutual exclusion on this internal state. Therefore, one task that is executing the random number generator can be pre-empted and the generator state can be modified by another task. This can result in problems with the generated random values or errors. One solution is to supply cover routines for each non-reentrant routine that guarantee mutual exclusion on calls, but this is not practical as too many cover routines have to be created.

Part of this problem can be handled by allowing pre-emption only in user code. When a pre-emption occurs, the handler for it checks if the current task is executing user code. If it is, the handler causes a context switch to another task. If the current task is not executing user code, the interrupt handler resets the timer and returns without rescheduling another task. In theory, a task that calls system routines at fortunate moments might never be pre-empted.

Determining whether an address is in user code is done in  $\mu C++$  by relying on the linker to place programs in memory in a particular order.  $\mu C++$  programs are compiled using a command that invokes the C++ compiler and includes all necessary include files and libraries. The command forces the linker to bracket all user modules between two precompiled routines, `uBeginUserCode` and `uEndUserCode`. The pre-emption interrupt handler simply checks if the interrupt address is between the addresses of `uBeginUserCode` and `uEndUserCode` to determine if the interrupt occurred in user code. This approach assumes that all libraries are non-pre-emptable, which inhibits concurrency for those routines that are reentrant (e.g. `sin`, `cos`, etc.).

Allowing pre-emption only in user code is sufficient to deal with non-reentrant routines on uniprocessors. On multiprocessors, we rely on the vendor to provide reentrant routines (which is not always a reasonable assumption). In the future, all library routines will have to be reentrant.

#### 4.3.1 I/O Libraries

The standard I/O libraries provide an example of undesirable interactions between libraries. To ensure maximum parallelism in light-weight tasking systems, it is desirable that a task not execute an operation that causes the processor it is executing on to block. UNIX I/O operations can be made to be nonblocking, but this requires special efforts since the I/O operations do not restart automatically when the operation completes. Instead, it is necessary to poll for I/O completions, and possibly block the program if all tasks are directly or indirectly blocked waiting for I/O operations to complete. Since this is complex, most concurrency libraries provide nonblocking versions of the I/O routines.

In  $\mu C++$ , I/O cover objects exist for the I/O streams, which check the ready queue before performing their corresponding C++ I/O operations. If no tasks are waiting to execute, blocking can occur because all tasks in the system must be directly or indirectly waiting for an I/O operation to complete. If tasks are waiting, a nonblocking I/O operation is performed. One of the tasks performing an I/O operation polls for completion of any I/O operation and yields control of the processor if no I/O operation has completed. When an I/O operation completes, all the I/O tasks are unblocked and each checks if its I/O operation has completed. This scheme allows other non-I/O tasks to make progress with only a slight degradation in performance due to the polling task.

## 4.4 Abnormal Event Handling

Handling abnormal events cannot be done properly using library mechanisms, such as return-codes, because these mechanisms do not scale to large robust systems. Virtually all new programming languages provide language facilities like exceptions to deal with abnormal situations. Concurrency adds another dimension to the handling of abnormal events in a program. How does exception handling work when there are multiple execution-states? How are hardware and software interrupt facilities introduced? In [BMZ], abnormal event handling mechanisms for concurrent environments were extensively analyzed. Two distinct mechanisms were identified:

1. An *exceptional* change in control flow, using the stack of an execution-state (i.e. C++ style exceptions).
2. A corrective action by an *intervention* in the normal computation of an operation, which includes interrupt/signal handling between tasks.

Facilities to support synchronous exceptions, and synchronous and asynchronous interventions were implemented through a library facilities in C. Unfortunately, the resulting library facilities burden users with both syntactic and semantic details, coding conventions and protocols. Our conclusion after constructing an abnormal event library is that it is impossible to provide powerful abnormal event handling mechanisms without augmenting the programming language. C++ was extended with exceptions for the same reason [KS90].

## 4.5 Task Libraries for Object-Oriented Languages

In an object-oriented language, the natural way to provide concurrency through a library is to define an abstract class, *Task*, that implements the task abstraction. The constructor for *Task* creates a thread to “animate” the task. User-defined task classes inherit from *Task*, and tasks are objects of these classes. This approach has been used to define C++ libraries that provide coroutine facilities [Sho87, Lab90] and simple parallel facilities [DG87, BLL88].

When this approach is used, task classes should have the same properties as other classes, so inheritance from task types should be allowed. Similarly, tasks should have the same properties as other objects. This latter requirement suggests that tasks should communicate via calls to member routines, since ordinary objects receive requests that way, and since the semantics of routine call matches the semantics of synchronous communication nicely. The body of the task (that is, the code that is executed by the thread associated with a task) has the job of choosing which member routine call should be executed next.

The following are the stages that a library package must deal with during the lifetime of a task:

1. thread creation for the task
2. task initialization
3. task body execution, which controls most of the synchronization with other tasks
4. task termination
5. joining/synchronization by another task with a task that is terminating

These stages will be referred to in the following sections.

### 4.5.1 Task Body Placement

The body of a task must have access to the members of a task, and the *Task* constructor must be able to find the body in order to start the task's thread running in it. Therefore, in the library approach, the task body must be a member of the task type. At first glance, the task's constructor seems like a reasonable choice. However, the requirement that it be possible to inherit from a task

type forbids that choice. Let T1 be a task type, with a constructor that contains initialization code for private data and the task body. Now consider a second type T2 that inherits from T1. T2's constructor must specify a new task body: it must somehow override the task body in T1's constructor, but still execute the private data initialization code. Given that they are contained in the same block of code, that is clearly impossible.

The correct solution is to put the body in a special member routine, perhaps called `main`. `main` would be declared by `Task`, and would be a virtual routine so that task types could replace it.

#### 4.5.2 Thread Creation

When one task creates another, the creating task's thread executes statements in `Task`'s constructor that create a new thread. The library implementor must decide which thread does what jobs. The approach that produces the greatest concurrency has the new thread execute the new task's constructors and body, while the creating thread returns immediately to the point of the declaration of the object. However, the normal implementation of constructors in C++ makes this difficult or impossible if inheritance from task types is allowed. Each constructor starts by calling the constructors of its parent classes. By the time `Task`'s constructor is called, there can be an arbitrary number of routine activations on the stack, one for each level of inheritance. It is not possible for the initialization code for `Task` to examine the stack to locate the return point for the original constructor. Only compiler support, such as marking the stack at the point of declaration or passing implicitly the return address for the creating thread up the inheritance chain, can make this approach work. In the absence of compiler support, the creating thread must execute the new task's constructors, while the new thread executes the task body, which inhibits concurrency somewhat.

#### 4.5.3 Task Initialization and Execution

The next problem results from an interaction between task initialization and task body execution. The task's thread must not begin to execute the task body until after the task's constructor has finished. However, in the library approach, the code to start the thread running in the task body appears in `Task`'s constructor. In C++, that code is executed first, *before* the constructors of any derived classes. Hence the new thread must be created in the "blocked" state, and must be unblocked after the derived constructors finish. A second, more subtle problem, results from the semantics of initialization. While `Task`'s constructor is executing, the new task is considered to be an instance of class `Task`, not the actual task class being instantiated. This means that, within `Task`'s constructor, the virtual `main` routine that contains the task's body is inaccessible; calling `main` in `Task`'s constructor will not execute the correct task body!

PRESTO dealt with this problem by requiring an explicit action to unblock the thread. In this approach, the `Task` class provides a `start()` member routine that must be called after the declaration of a task, but before any calls to the task's member routines. At that point the constructors have all finished and `main` refers to the actual task body. This two-step creation protocol opens a window for errors: programmers may fail to start their tasks.

A similar interaction exists between task body execution and task termination. When one task deletes another, it will call the deleted task's destructor. The destructor must not begin execution until after the task body has finished. However, the code that waits for the task body to finish cannot be placed in `Task`'s destructor, because it would be executed last, *after* the destructors of any derived classes. Task designers cannot simply move the task's termination code from the destructors to the end of the task body, because that would prevent further inheritance: derived classes would have no way to execute their base class's termination code. `Task` could provide a `finish()` routine, analogous to `start()`, which must be called before task deletion, but this two-step termination protocol is even more error-prone than the creation protocol.

A general language mechanism like Simula's `inner` [Sta87] would solve these problems. In a single inheritance hierarchy, an `inner` statement in a constructor (or destructor) of a base class acts like a call to the constructor (or destructor) of the derived class. For instance, given

```

class T1 {
public:
    T1() { s1; inner; s2; };
};
class T2: public T1 {
public:
    T2():T1() { s3; };
};
T1 a_t1;
T2 a_t2;

```

the initialization of `a_t1` executes statements `s1` and `s2`, and the initialization of `a_t2` executes `s1`, `s3`, and `s2`, in that order. (`T2::T2` might also contain inner statements, which would invoke the constructors of classes derived from `T2`.) In `T1::T1`, before the inner statement, `a_t2` is considered to be an instance of class `T1`. After the inner statement, it is an instance of `T2`, and the meaning of calls to virtual routines changes accordingly.

A concurrency library would use `inner` in `Task` to control the timing of events. `Task`'s constructor would use an inner statement to execute the derived task class's constructors and establish the proper meaning for `main`, and then create the new thread running (unblocked) in `main`. `Task`'s destructor would wait for the task body to finish, and then would use `inner` to execute the task class's destructors. However, with this technique the creating thread executes the constructors, which inhibits concurrency as mentioned in section 4.5.2.

The inner statement is useful, independent of concurrency, for fine control during the initialization of an object. For instance, it lets constructors of base classes call virtual functions that are redefined by derived classes. However, it is not obvious how `inner` could be added to C++. `inner` must invoke constructors and destructors in the order defined by C++, taking multiple inheritance and virtual inheritance into account. Furthermore, a class can have many constructors, and descendants specify which of their base class's constructors are called. Finally, there should be some canonical translation from `inner` to efficient, standard C.

#### 4.5.4 Task Communication

Communication among tasks also presents difficulties. In library-based schemes (and some languages), it is done via message queues, called ports or channels [And91]. However, a single queue per task is inadequate; the queue's message type inevitably becomes a union of several "real" message types, and static type checking is compromised. Inheritance from a `Message` class could be used, instead of a union, but the task would then have to perform type tests on messages before accessing them with facilities like Simula's `is` and `qua`. Currently, runtime type-tests are counter to the C++ design philosophy.

If multiple queues are used, a library facility analogous to the Ada [Uni83] `select` statement is needed to allow a task to wait for messages to arrive on more than one queue. However, building a library facility similar to a `select` statement requires  $\lambda$ -expressions (anonymous nested routine bodies) or preprocessor macros to support the blocks of code that may or may not be invoked depending on the selection criteria, for example:

```

select( accept( queue-name, code-body ) || accept( ..., ... ) || ... );

```

where the *code-body* is a  $\lambda$ -expression and represents the code executed after a particular message queue is accepted. This capability is essential so that a particular action can be performed after a message is received. Furthermore, there is no statically enforceable way to ensure that only one task is entitled to receive messages from any particular queue, for example:

```

MsgQueueType A;
MsgQueueType B;

class TaskType : public Task {
    void main() {          // task body
        ...
        select( accept( A, NULL ) || accept( B, NULL ) );
        ...
    }
};

TaskType T1, T2;

```

Tasks T1 and T2 simultaneously accept messages from the same queues. While it is straightforward to check for the existence of data in the queues, if there is no data, both T1 and T2 must wait for data to appear on either queue. To implement this, tasks have to be associated with both queues until data arrives, given data when it arrives, and then removed from both queues. This implementation would be expensive since the addition or removal of a message from a queue would have to be an atomic operation across all queues involved in a waiting task's accept statement to ensure that only one data item from the accepted set of queues is given to the accepting task. In languages with concurrency support, the compiler can disallow accepting from overlapping sets of message queues by restricting the select statement to queues the task declares. Compilers for more permissive languages, like SR [AOC<sup>+</sup>88], perform global analysis to determine if tasks are receiving from overlapping sets of message queues; in the cases where there is no overlap, less expensive code can be generated. In a library approach, access to the message queues must assume the worst case scenario.

If the routine-call mechanism is to be used for communication among tasks (as in  $\mu$ C++), a select statement again requires a  $\lambda$ -expressions or preprocessor macros. Furthermore, each public member routine has to have special code at the start and possibly at the exits, which the programmer has to provide by following a convention. This special code would provide, at the least, mutual exclusion and control selective entry. Object-oriented programming languages that support inheritance of routines, such as LOGLAN'88 [CKL<sup>+</sup>88] and Beta [KMMPN87], can provide special member code automatically. (The use of inner in a constructor is a special case of routine inheritance, where the derived class's constructor inherits from the base class's constructor.) Whatever the mechanism, it must allow the special code to be selectively applied to the member routines. For example, there are cases where not all public member routines require mutual exclusion and where some private members require mutual exclusion. In languages with concurrency support, the compiler can easily disallow accepting another task's member, so the problem of accepting from overlapping sets of members will not occur.

## 4.6 More Inheritance Problems

Regardless of whether a concurrency library or language extensions are used to provide concurrency in an object-oriented language, new kinds of types are introduced, like coroutine, monitor, and task. These new kinds of types complicate inheritance. The trivial case of single inheritance among homogeneous kinds, i.e. a monitor inheriting from another monitor, is straightforward because any implicit actions are the same throughout the hierarchy. (An additional requirement exists for tasks: there must be at least one task body specified in the hierarchy.) For a task or a monitor type, new member routines that are defined by the derived class can be accepted by statements in a new task body or in redefined virtual routines.

Inheritance among heterogeneous types can be both useful and confusing. Having classes with mutual exclusion inherit from classes without it is useful to generate concurrent types from existing non-concurrent types. For instance, a sharable queue task could be defined by inheriting from an ordinary queue and redefining all of the class's member routines to provide mutual exclusion.



```

class Queue {
public:
    void insert( ... ) ...
    virtual void remove( ... ) ...
};
class MutexQueue : public Queue, public Task {
    virtual void insert( ... ) ...
    virtual void remove( ... ) ...
};

```

However, this example demonstrates the dangers caused by non-virtual routines.

```

Queue *qp = new MutexQueue;    // subtyping allows assignment
qp->insert( ... );              // call to a non-virtual member routine, statically bound
qp->remove( ... );              // call to a virtual member routine, dynamically bound

```

Queue::insert does not provide mutual exclusion because it is a member of Queue, while MutexQueue::insert and MutexQueue::remove do provide mutual exclusion. Because the pointer variable qp is of type Queue, the call qp->insert calls Queue::insert even though insert was redefined in MutexQueue; no mutual exclusion occurs. In contrast, the call to remove is dynamically bound, so the redefined routine in the monitor is invoked and appropriate synchronization occurs. The unexpected lack of mutual exclusion would cause many errors. In object-oriented programming languages that have only virtual member routines, this is not a problem. The problem does not occur with C++'s private inheritance because no subtype relationship is created and hence the assignment to qp would be invalid.

Heterogeneous inheritance among entities like monitors, coroutines and tasks can be very confusing. While it is always possible to construct some meaning for such inheritance, we reject it for the following reason. Classes are written as ordinary classes, coroutines, monitors, or tasks, and we do not believe that the coding styles used in each can be arbitrarily mixed. For example, an instance of a task class that inherits from an ordinary class can be passed to a routine expecting instances of the class. If the routine calls one of the object's member routines, it could inadvertently block the current thread indefinitely. While this could happen in general, we believe there is a significantly greater chance if users casually combine types of different kinds.

Multiple inheritance simply exacerbates the problem stated above and it significantly complicates the implementation, which slows the execution. For example, accepting member routines is significantly more complex with multiple inheritance because it is not possible to build a static mask to test on routine entry. As is being discovered, multiple inheritance is not as useful a mechanism as it initially seemed [BCK89, Car90].

## 4.7 Libraries for Non-Object-Oriented Languages

What are the problems of adding concurrency to non-object-oriented languages and can it be done using a library approach?

We have extensive experience in adding concurrency to C using the library approach in a system called the  $\mu$ System [BS90]. The  $\mu$ System is a light-weight tasking library for C that runs on the following processors: M68K, NS32K, VAX, MIPS, i386/486, Sparc, and the following UNIX operating systems: Apollo SR10 BSD, Sun OS 4.x, Tahoe BSD 4.3, Ultrix 3.x/4.x, DYNIX, Umax 4.3, IRIX 3.3. As well, the  $\mu$ System provides coroutines, several forms of synchronization and communication, and synchronous and asynchronous abnormal event handling features. We can say with authority that many of the design criteria stated earlier cannot be provided without language extensions. This statement is true for all light-weight tasking systems for C using a library approach [Che82, Gen85, Che88, Sun88, Enc88, CG89]. Only because C allows extensive violations of its type system is it possible to build an adequate set of library facilities. In general, either the type system is violated or the functionality is restricted, as illustrated in the following examples. In the library approach, a task is formed by starting a thread executing in a routine body. If this routine is



allowed to have arbitrary parameters, there is no type safe way to pass arguments in a library approach. If the routine is not allowed to have parameters or has a fixed parameter list, this complicates initialization as a protocol is now required between the creator and the new task to pass the initialization values that cannot be provided when the task is started. Type-safe direct communication among tasks is impossible in a library approach because a routine has only one entry point that can be invoked in a type safe way. Type-safe indirect communication is possible through a monitor library, but a library monitor requires user conventions for proper usage. These problems exist in thread packages in other languages, such as Modula-2.

Non-object-oriented languages with language support for concurrency [MMS79, HC88] can pass initial arguments to a new task in a type-safe way because a special statement to start the new task is provided. However, these languages cannot solve the direct communication problem, and hence, do not provide direct communication. Communication is normally indirect through monitors, which may be part of the language so that user conventions are unnecessary.

In [Hil83, pp. 136–144], Hilfinger outlines how concurrency might have been added to the programming language Ada using elementary programming language constructs instead of high-level programming language constructs (Ada's existing concurrency facilities are largely class-based). In Hilfinger's proposal, creating threads and execution-states are still elementary properties [Hil83, pp. 141–142] that are tied into the language's runtime environment at a very low-level.  $\lambda$ -expressions and routine variables (pointers to routines), which Ada does not have, are required to build a select statement. Furthermore, without automatic dereferencing of pointer variables, usage syntax would be unacceptable. Finally, users are required to follow coding conventions for each routine that requires mutual exclusion. Inheritance problems do not exist because Ada's type system does not have inheritance. (The Ada 9X proposals has a large number of new language features to support concurrency rather than building on existing language features.) With all its faults, Hilfinger's outline does support type-safe direct communication. Appendix A presents our minimalistic approach for including concurrency into a non-object-oriented language that supports type-safe direct communication. Our solution requires language support to achieve all the design options stated earlier.

## 5 Conclusion

Our main conclusion is that concurrency is a fundamental aspect of a programming language that cannot be built easily from primitive non-concurrent language constructs. Creation of a thread and execution-state cannot be implemented from basic constructs without working at a very low-level, possibly violating both type-safety and the integrity of the runtime environment, nor can mutual exclusion be implemented inexpensively. Even given the three elementary properties, library facilities do not usually integrate into the language's type system, often requiring protocols and coding conventions, and library facilities may be inefficient. The purpose of high-level languages is to automate protocols and conventions, and provide global optimization to make a program efficient.

The following are some specific observations:

- If a language supports all the design options presented at the beginning, a large number of different models of concurrency can be implemented. Usually, the expressive power of the language is the limiting factor as to how well a model can be expressed.
- Minimizing the cost of a context switch requires language support because only the compiler knows exactly how much state a particular object is using.
- Without language support, object-oriented languages with inheritance have problems in coordinating initialization and the starting of a task's new thread, as well as specifying the location where the thread starts execution.
- Indirect type-safe communication using message queues, where selection can occur from multiple overlapping message queues, is costly to implement.

- Direct type-safe communication requires an aggregating construct with multiple entry points. Therefore, class-based programming languages appear to be able to support direct type-safe communication better than non-object-oriented languages. This advantage results from the special relationship between the class and its member routines, which can be extended with other properties like mutual exclusion.
- If an object-oriented language provides special type-specifiers, like task and monitor, there are addition problems with heterogeneous inheritance among the type specifiers, e.g. a class which inherits from a monitor which inherits from a task.
- A language's exception handling mechanism must be designed to work with its concurrency mechanism because exceptions work with the stack associated with an execution-state (exceptions search the execution stack) and there are now multiple execution-states. Furthermore, a mechanism to deal with interrupts must be provided.

## A Concurrency in Non-Object-Oriented Programming Languages

In non-object-oriented languages, data structures corresponding to objects are created by instantiation of pure types (i.e. types containing only data fields). These types are grouped together with the routines that manipulate the data structures into collections such as modules; we refer to these languages as routine/type/module (RTM) languages (e.g. Ada, Modula-2 [Wir85]). Since a module does not define a type, subtyping is not a concept that pertains to modules. Instead, polymorphism/reuse is accomplished through overloading, parameter generalization, and call-site inferencing and binding [CW90]. Data structures in RTM languages can be manipulated like objects using the following conventions. A module exports an opaque type, which corresponds to the class type. The opaque type provides encapsulation and information hiding outside the module but the module routines can access all the fields of the type for instances passed as arguments. Initialization and termination code can be associated with an opaque type by overloading the routines *new* and *delete*, which are called implicitly after allocation and before deallocation of a data item. (We do not know of any RTM language that supports this facility, but it is possible if the language's polymorphism capabilities permit overloading.) Therefore, any class can be transformed into an opaque type in a module.

Before discussing how concurrency can be added to RTM languages, we want to dismiss approaches that use modules to mimic certain kinds of objects, e.g. monitor modules [MMS79, HC88], as they are not general. In this approach, a module creates a single object and for a task, execution would begin in the module initialization code. However, since a module is instantiated only once, it does not allow creation of multiple instances of the object. Generic modules can mitigate this problem, but we feel that using generics to generate multiple instances of the same type is an inappropriate use of this facility.

When opaque types are used to define classes, calls to module routines are normally unsynchronized, that is, the caller's state is saved and control transfers to the called routine. However, this is not the case for routines that operate on monitor or task types (or persistent types [BZ88]). Special call-action, namely that required to implement mutual exclusion and/or task synchronization, must be provided by routines that directly access the fields of these types. Furthermore, creation of instances of the module's opaque type may also have to start a new thread of control at a particular location. When a language provides special constructs, e.g. task construct, the special call-action is implied; however, it must be explicitly stated when using RTM languages.

Figure 1 shows a possible implementation of a bounded buffer as a class-based task and a RTM task. The implementation of the RTM task preserves all the initial design options so the comparison is fair. First, a task record's semantics are that instantiation creates a new thread and the thread begins execution in the overloaded routine *new* that has a parameter of the corresponding record type. The opaque facility provides information hiding capability. (The overloaded routine *delete* would be called implicitly to perform termination after the task's thread has terminated and before deallocation.) Second, a routine with a task record parameter implies that a task making a call to it

Class-based Task	RTM-based Task
<pre> task buffer {     const int QSize = 3;      int front, back, count;     int queue[QSize];  protected:     void main() {         front = back = count = 0;          for ( ;; ) {             accept( stop )             break;             or when (count != QSize)                 accept( insert );             or when (count != 0)                 accept( remove );         }; // for     }; // main public:     void stop() {     }; // stop      void insert(int elem) {         queue[back] = elem;         back = (back + 1) % QSize;         count += 1;     }; // insert      int remove() {         int elem;          elem = queue[front];         front = (front + 1) % QSize;         count -= 1;         return(elem);     }; // remove }; // buffer  buffer a, b, c;  a.insert(1); i = a.remove(); </pre>	<pre> module buffermodule {     export buffer, new, stop, insert, remove;     const int QSize = 3;     opaque type buffer = task record {         int front, back, count;         int queue[QSize];     }; // buffer      void new(buffer b) {         b.front = b.back = b.count = 0;          for ( ;; ) {             accept( stop )             break;             or when (count != QSize)                 accept( insert );             or when (count != 0)                 accept( remove );         }; // for     }; // new      void stop(buffer b) {     }; // stop      void insert(buffer b, int elem) {         b.queue[back] = elem;         b.back = (b.back + 1) % QSize;         count += 1;     }; // insert      int remove(buffer b) {         int elem;          elem = b.queue[front];         front = (front + 1) % QSize;         count -= 1;         return(elem);     }; // remove }; // buffermodule  buffer a, b, c;  insert(a, 1); i = remove(a); </pre>

Figure 1: Bounded Buffer using Class-based Task and RTM-based Task

is blocked if the corresponding argument is currently being accessed by another task. In that case, the caller must be put on one of a number of hidden queues associated with the argument corresponding to that task record parameter; this would result in an efficient implementation. When the routine finishes, the task argument is released and is available for access by other tasks. A routine can only have one parameter that requires mutual exclusion, since putting a task on multiple queues does not make sense. Because of this restriction, having multiple task records defined in a single module would be done solely for organizational reasons as no routine could access both task records. If an exported module routine simply wants to pass the task record parameter on to another routine without requiring mutual exclusion, it must qualify the parameter type with a `nomutex` qualifier. Lastly, our example allows a task to be blocked using `wait` and `signal` statements and condition variables as for a task created using a class-based task.

Any routine written outside of the module that has a parameter of a module's task-record type would not acquire mutual exclusion (the parameter would be implicitly `nomutex`). The following problems arise if this rule is not adopted. First, the implementation of a task record could not use multiple entry queues because a new queue would have to be added to the task-record type for the new routine and this would require extending the record, which cannot be done. At best, a single queue must be used and it must be searched when an accept is executed; this might substantially affect performance. Second, the new routine cannot access any of the opaque fields of the task data structure, except indirectly through the routines provided in the module, hence there is little point in obtaining mutual exclusion. As a result, parametric polymorphism techniques cannot be used to provide code reuse.

If an RTM language supported record concatenation [Wir88] that is applicable to task-record types, this being analogous to object-oriented inheritance, this would result in equivalent implementation with a class-based task, and hence, the same performance. In this case, a new module would define a task-record type that is an extension of an existing task-record type from another module. For each routine in the new module with a task record parameter of the extended task-record type, a new entry queue would be added to the extended type.

We believe that this outline for RTM tasks is probably the best way to extend RTM languages to support types with special semantics. The drawback is that special semantics actions must be explicitly specified, e.g. `nomutex` clause, by the user and its implementation might imply some restrictions on the polymorphism mechanism or vice versa.

## References

- [ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53-79, February 1992.
- [Agh86] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.
- [And91] Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49-90, March 1991.
- [AOC+88] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An Overview of the SR Language and Implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51-86, January 1988.
- [BCK89] Harry Bretthauer, Thomas Christaller, and Jürgen Kopp. Multiple vs. Single Inheritance in Object-oriented Programming Languages. What do we really want? Technical Report Arbeitspapiere der GMD 415, Gesellschaft Für Mathematik und Datenverarbeitung mbH, Schloß Birlinghoven, Postfach 12 40, D-5205 Sankt Augustin 1, Deutschland, November 1989.

- [BDS<sup>+</sup>92] P. A. Buhr, Glen Ditchfield, R. A. Strooboscher, B. M. Younger, and C. R. Zarnke.  $\mu$ C++: Concurrency in the Object-Oriented Language C++. *Software-Practice and Experience*, 22(2):137-172, February 1992.
- [BLL88] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-oriented Parallel Programming. *Software-Practice and Experience*, 18(8):713-732, August 1988.
- [BMZ] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and Asynchronous Handling of Abnormal Events in the  $\mu$ System. to appear in *Software-Practice and Experience*.
- [BS90] Peter A. Buhr and Richard A. Strooboscher. The  $\mu$ System: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software-Practice and Experience*, 20(9):929-963, September 1990.
- [BS92] Peter A. Buhr and Richard A. Strooboscher.  $\mu$ C++ Reference Manual, Version 3.4.2. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, June 1992.
- [BW88] Hans-J. Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software-Practice and Experience*, 18(9):807-820, September 1988.
- [BZ88] P. A. Buhr and C. R. Zarnke. Nesting in an Object Oriented Language is NOT for the Birds. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object Oriented Programming*, volume 322, pages 128-145, Oslo, Norway, August 1988. ECOOP'88, Springer-Verlag. Lecture Notes in Computer Science, Ed. by G. Goos and J. Hartmanis.
- [Cah] Conor P. Cahill. debug\_malloc. comp.sources.unix, volume 22, issue 112.
- [Car90] T. A. Cargill. Does C++ Really Need Multiple Inheritance? In *USENIX C++ Conference Proceedings*, pages 315-323, San Francisco, California, U.S.A, April 1990. USENIX Association.
- [CDG<sup>+</sup>88] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 Report. Technical Report 31, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, August 1988.
- [CG89] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444-458, April 1989.
- [Che82] D. R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability*. American Elsevier, 1982.
- [Che88] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314-333, March 1988.
- [CKL<sup>+</sup>88] Boleslaw Ciesielski, Antoni Kreczmar, Marek Lao, Andrzej Litwiniuk, Teresa Przytycka, Andrzej Salwicki, Jolanta Warpechowska, Marek Warpechowski, Andrzej Szalas, and Danuta Szczepanska-Wasersztrum. Report on the Programming Language LOGLAN'88. Technical report, Institute of Informatics, University of Warsaw, Pkin 8th Floor, 00-901 Warsaw, Poland, December 1988.
- [CW90] G. V. Cormack and A. K. Wright. Type-dependent Parameter Inference. *SIGPLAN Notices*, 25(6):127-136, June 1990. Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation June 20-22, 1990, White Plains, New York, U.S.A.

- [DG87] Thomas W. Doeppner and Alan J. Gebele. C++ on a Parallel Machine. In *Proceedings and Additional Papers C++ Workshop*, pages 94–107, Santa Fe, New Mexico, U.S.A, November 1987. USENIX Association.
- [Enc88] Encore Computer Corporation. *Encore Parallel Thread Manual*, 724-06210, May 1988.
- [ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, first edition, 1990.
- [Gen81] W. Morven Gentleman. Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept. *Software-Practice and Experience*, 11(5):435–466, May 1981.
- [Gen85] W. Morven Gentleman. Using the Harmony Operating System. Technical Report 24685, National Research Council of Canada, Ottawa, Canada, May 1985.
- [GR89] N. H. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, Summit, NJ, 1989.
- [HC88] R. C. Holt and J. R. Cordy. The Turing Programming Language. *Communications of the ACM*, 31(12):1410–1423, December 1988.
- [HD90] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. *SIGPLAN Notices*, 25(3):128–136, March 1990. Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, March. 14–16, 1990, Seattle, Washington, U.S.A.
- [Hil83] Paul N. Hilfinger. *Abstraction Mechanisms and Language Design*. ACM Distinguished Dissertations. MIT Press, 1983.
- [Hoa73] C. A. R. Hoare. Hints on Programming Language Design. Technical Report CS-73-403, Stanford University Computer Science Department, December 1973. Reprinted in [Was80].
- [JW85] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, third edition, 1985. Revised by Andrew B. Mickel and James F. Miner, ISO Pascal Standard.
- [KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA Programming Language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, Computer Systems Series, pages 7–48. MIT Press, 1987.
- [KS90] Andrew Koenig and Bjarne Stroustrup. Exception Handling in C++. *Journal of Object-Oriented Programming*, 3(2):16–33, July/August 1990.
- [Lab90] Pierre Labrèche. Interactors: A Real-Time Executive with Multiparty Interactions in C++. *SIGPLAN Notices*, 25(4):20–32, April 1990.
- [MMS79] James G. Mitchell, William Maybury, and Richard Sweet. Mesa Language Manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.
- [Sho87] Jonathan E. Shopiro. Extending the C++ Task System for Real-Time Control. In *Proceedings and Additional Papers C++ Workshop*, pages 77–94, Santa Fe, New Mexico, U.S.A, November 1987. USENIX Association.
- [Sta87] Standardiseringskommissionen i Sverige. *Databehandling – Programspråk – SIMULA*, 1987. Svensk Standard SS 63 61 14.

- [Sun88] *System Services Overview, Lightweight Processes*, chapter 6, pages 71–111. Sun Microsystems, May 1988. available as Part Number: 800-1753-10.
- [Uni83] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Published by Springer-Verlag.
- [Was80] Anthony I. Wasserman, editor. *Tutorial: Programming Language Design*. Computer Society Press, 1980.
- [Wir74] Niklaus Wirth. On the Design of Programming Languages. In *Information Processing 74*, pages 386–393. North Holland Publishing Company, 1974. Reprinted in [Was80].
- [Wir85] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, third, corrected edition, 1985.
- [Wir88] N. Wirth. Type Extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.
- [ZH88] Benjamin Zorn and Paul Hilfinger. A Memory Allocation Profiler for C and Lisp Programs. In *Summer 1988 USENIX proceedings*, 1988.





# A Portable Implementation of C++ Exception Handling

Don Cameron

Paul Faust

Dmitry Lenkov

Michey Mehta

*Hewlett-Packard California Language Laboratory*

*E-Mail: {dcc|pfaust|dmitry|mnm}@cup.hp.com*

## ABSTRACT

We have designed and implemented a portable implementation of C++ exception handling. We use the word *portable* because we translate C++ code into C code which can be compiled on any system which provides a C compiler. We chose to design a portable implementation because exception handling is a new language feature, and it is important to provide an early implementation on a variety of platforms. We discuss the implementation of transfer of control, exception identification, object cleanup, and run-time storage management. In addition, we mention some aspects of the C++ exception handling language specification which needed clarification during our implementation. Finally, we describe the run-time performance of our implementation, and show how a compiler that produces object code directly can provide better performance.

## 1. Introduction

We assume the reader is familiar with the C++ Exception Handling language definition as described in references [1], [2], and [3]. We have modified a *cfront* 3.0 translator to support C++ exception handling, and this section is an implementation overview. Our implementation can be broken down into four functional areas:

### Transfer of Control

When an exception is encountered, the exception handling mechanism must suspend execution at the throw point, and resume execution at the appropriate catch point. When execution is resumed, global and local variables must have correct values. The "Transfer of Control" section describes the mechanism used to determine where execution should be transferred to, and describes the mechanism used to transfer control to the chosen destination.

### Exception Identification

The exception handling run-time support (henceforth, simply the "run-time") must have type information available which describes various characteristics of a type; for example, this information is used to determine if a thrown exception is handled by a catch clause. The mechanism for emitting and utilizing this information is called "Exception Identification".

### Object Cleanup

When an exception occurs, the exception handling mechanism should attempt to destroy all fully and partially constructed automatic objects between the throw point and the catch point. If an exception occurs in the construction of a heap object, the heap object should be destroyed and any memory allocated for the object should be deallocated. When `exit` is called, fully and partially constructed static objects should be destroyed. The mechanism for handling these actions is described in the "Object Cleanup" section.

### Storage Management

The run-time must maintain a copy of a thrown object. There can be multiple thrown objects which are simultaneously active, and the run-time must manage the memory necessary to store such objects. The mechanism for implementing this functionality is described in the "Storage Management" section.

Our portable exception handling scheme can be summarized as follows:

- We use `setjmp/longjmp` to transfer control from a **throw** to the appropriate **catch** clause.
- We use a linked list of "markers" running through the stack to record the execution of try blocks, functions with exception specifications, and functions which require object cleanup.
- The translator emits **typeinfo** objects to store useful information about a type (such as the list of base classes). This information is used by the exception mechanism to determine if a catch clause can handle the thrown object, and to check for exception specification violations. We also use the **typeinfo** information to determine how to destroy partially constructed objects.
- Upon entry into a function which requires object cleanup in the event of an exception, we chain a "cleanup marker" into the chain of markers. This cleanup marker will point to a statically generated table which describes the cleanup actions required by this function.
- The chain of markers is also used to handle functions with exception specifications; this is done by adding a "specification marker" to the marker chain upon entry to a function with an exception specification.

## 2. Transfer of Control

We implement portable transfer of control using the standard C library routines *setjmp* and *longjmp*, which provide a nonlocal goto facility. In this section we illustrate our implementation of transfer of control with a fragment of C++ code, and a representation of the equivalent C code generated by the translator. We also discuss the actions taken by the run-time when an exception is thrown.

---

```
int j;
try {
    j = 100;
    f();
}
catch ( String obj ) {
    obj.len = 99;
}
catch ( int& obj2 ) {
    obj2 = 100;
}
```

---

Figure 1. C++ Source

Figure 1 shows a fragment of C++ code which contains a **try** block. The translated pseudo C code for this fragment is shown in figure 2. Code for object cleanup and for declaration of some compiler-generated types is omitted.

Executing the **try** keyword is realized by

- chaining a *try marker* into a global chain accessible to the run-time.
- executing an "if `setjmp ...`" which initializes a `setjmp` buffer in the try marker.

Following the "execution" of the **try** keyword, the code contained in the try block is executed. If no exception is thrown, the catch clauses (handlers) are skipped, and the try marker is unchained.

If an exception is thrown, a run-time routine is called with a copy of the thrown object and a representation of its type. This routine makes an initial pass over the chain of markers, comparing the thrown object's type with the types of handlers associated with try markers on the list. If no matching handler is found, the function `terminate()` is called. In the course of walking the list, if an exception specification violation is found, `unexpected()` is called. Returning from either routine leads

to a call of `abort()`.

Try markers on the list can be in one of three states. An *available* marker represents execution in a try block, and is a candidate to handle the current exception. A *busy\_in\_handler* marker represents execution in a handler, and is not a candidate to handle another exception. For try markers which are *busy\_doing\_cleanup*, see below.

If a match is found with the current marker, the run-time throw routine performs the following actions:

1. The try marker associated with the matching catch clause is marked as *busy\_doing\_cleanup*. This allows the run-time to detect if a destructor invoked for cleanup attempts to exit by throwing an exception.
2. A second pass over the chain of markers is performed from the head of the list to the destination try marker. As cleanup markers are encountered, objects in activations which will disappear as the stack is cut back are cleaned up. Finally, objects in the try block itself are cleaned up.

A *busy\_in\_handler* marker which will be popped off the marker list represents a handler in execution, and contains a copy of a previously thrown object. The object's destructor must be executed. In addition, storage which was allocated by run-time support for a copy of the thrown object must now be deallocated.

3. The "catch\_index" in the try marker is set to select the matching catch clause.
4. The head of the marker list is set to point to the current marker (the marker "handling" the exception).
5. The marker state is set to *busy\_in\_handler*.
6. A `longjmp` is performed using the marker's `setjmp` buffer.

## 2.1 Optimization of Try Blocks

The `setjmp/longjmp` facility does not guarantee that non-volatile auto locals modified after a call to `setjmp` will have their new values restored when a corresponding call to `longjmp` occurs. This limitation does not cause problems for unoptimized code, where auto variables are typically updated in memory immediately. However, optimization can cause auto locals to be promoted to registers, and an optimizer will not be aware that there is, in effect, a path from any call in a try block to the `else` clause representing the handler list.

The ANSI C `volatile` type specifier, if applied to variables modified in a try block, ensures that memory is kept up to date. The use of `volatile` is undesirable in our portable implementation for two reasons:

- It is overkill. The variables in question need only be in memory at call points, not everywhere in the try block.
- The `volatile` facility is not supported by many non-ANSI C compilers.

It is possible to fool a conventional global optimizer into thinking that auto locals are globally readable and modifiable. Thus auto locals will have their memory locations updated before any call (including the call to the throw routine), and they will be reloaded from memory after a call.

The addresses of any locals modified in the try block are passed to the dummy routine "fake\_volatile", ensuring that memory for modified auto variables is kept up to date. To avoid the run-time overhead of executing the call, we place it in an unreachable compiler-generated handler. This handler includes a `goto` back to the start of the try, ensuring that affected locals' addresses appear to be globally known beginning with the try.

---

```

{
    // A block is introduced to contain a try marker and catch clause info.
    catch_table CT = { /* info about catch clauses */ };
    // A try marker is declared and pushed onto the global chain.
    try_marker try_object;
    try_object.previous = marker_head; marker_head = & try_object;
    try_object.state = available;
    try_object.info_ptr = &CT;
    int j; // Declare auto local.
    // Translation of the try block:
    if ( setjmp( & try_object.setjmp_buf ) == 0 ) {
        try_start :
            j = 100;
            f();
    }
    else { // The catch clauses are contained within this else
        if (try_object.eh_index == 1) {
            String obj;
            // Call copy constructor since obj is caught by value
            String_copy_ctor(& obj, try_object.caught_obj_addr);
            obj.len = 99;
        }
        else if (try_object.eh_index == 2) {
            // Allocate a pointer since obj2 is caught by reference
            int* obj2 = try_object.caught_obj_addr;
            *obj2 = 100;
        }
        else { // Ensure locals modified in try block are kept in memory.
            // This else clause should never be reached.
            fake_volatile( & j );
            goto try_start;
        }
    }
    // Unchain the marker and deallocate thrown object if any
    marker_head = try_object.previous;
    if (try_object.state == busy_in_handler) dealloc(& try_object);
}

```

---

**Figure 2.** Translator Pseudo Output

### 3. Exception Identification

When an exception is thrown, we need some mechanism to represent the type of the exception, so that the run-time can match the thrown exception with the appropriate handler. This mechanism is also used to check exception specifications and to provide information for performing object cleanup. We call this mechanism "exception identification", and this section describes our implementation.

Our compiler creates a run-time representation of the class hierarchy, for these reasons:

- The run-time uses this hierarchy to determine if the type mentioned in a catch clause is an accessible base of the thrown type.
- As we show in the section on "Object Cleanup", if the run-time knows the class hierarchy of a partially constructed object, it can use this information to destroy all fully constructed sub-objects when unwinding the stack. Without this class hierarchy, we would have to generate code to leave

an audit trail containing enough information for the run-time to destroy a partially constructed object in the event of an exception; the run-time cost for this is prohibitive.

The design presented below attempts to minimize the space used by **typeinfo** information, while allowing the run-time to make subtype inquiries efficiently.

### 3.1 Implementation Overview

An exception type will be represented by a **typeinfo** object. The **typeinfo** object for a class (Figure 3) will contain at least the following information: a list of base classes, the visibility of each base class, an indication of whether each base class is virtual or not, the destructor count for the class, and the offset of each base class within the derived class. The offset information is needed to convert a derived class to a base class. The **typeinfo** object for all non-class types will be objects which contain no information; these objects will only be used for address comparisons. One exception to the previous rule is made for pointers and references to classes, which will use the exception id of the class (refer to the section "Pointers and References to Classes"). Figure 3 is a pictorial representation of a **typeinfo** object for classes.

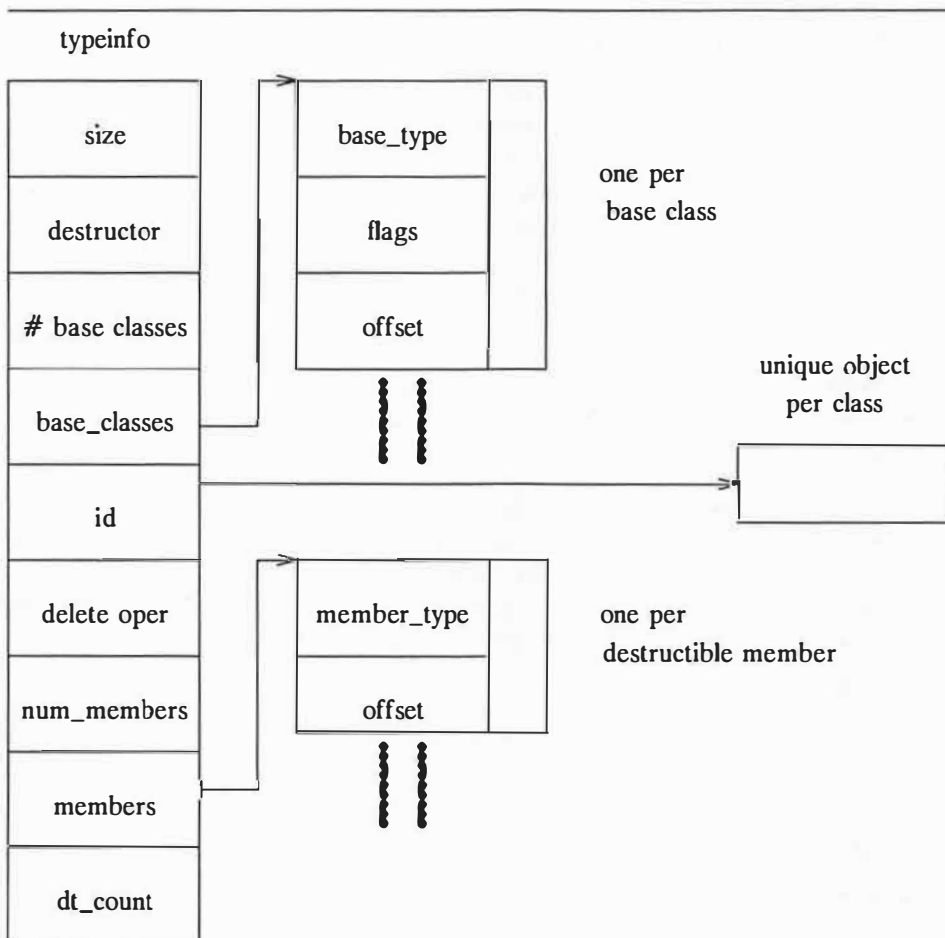


Figure 3. Structure of a **typeinfo** object

Allocating a **typeinfo** object for *every* type encountered in a program is not necessary, since a compiler can determine whether or not the **typeinfo** object for a type can be accessed at run-time. **typeinfo** objects can be referenced at run-time for any of the following reasons:

1. We must have **typeinfo** objects for the static types of any types thrown. For types which are classes, we must also allocate **typeinfo** objects for the ancestor class hierarchy. This is used by the exception handling mechanism to determine if a catch clause for a base class can catch a throw of a derived class object.
2. We must have a **typeinfo** object for every type mentioned in a catch clause, since a thrown exception must be compared against the type in a catch clause to see if it is handled.
3. We allocate a **typeinfo** object for the type of all destructible objects to support object cleanup.
4. We allocate a **typeinfo** object for every type which appears in a throw specification.

The separate compilation model of C++ makes it difficult to always emit only a single **typeinfo** object for each type. A close approximation can be made by using the same algorithm as *cfront* uses for the emission of virtual tables (a virtual table for a class is generated in the compilation unit which defines the first non-inline virtual function). When a unique **typeinfo** object cannot be emitted for a class, we allocate **typeinfo** objects in every compilation unit which requires one; this slightly complicates determining type equivalence at run-time (see the section entitled "Determining **typeinfo** Equivalence").

### 3.2 Pointers and References to Classes

Pointers and references to classes do not really need their own **typeinfo** objects, but instead use the **typeinfo** object of the pointed to class. When a pointer to a class is thrown, we pass the **typeinfo** object of the class as a parameter to the throw routine, along with an additional parameter which indicates that a pointer was thrown. When a pointer or reference to a class is caught, the catch table associated with the try block will indicate the **typeinfo** object of the class, along with a flag that indicates if this class was caught by pointer or reference. This information is sufficient to allow the exception handling mechanism to match the appropriate catch clause with the thrown exception.

### 3.3 Non-Class Types

The run-time only needs to determine if two non-class types are the same or not; no non-trivial conversions are performed at run-time. Each non-class type (such as `int`, `String**`, etc.) will have a unique **typeinfo** object because these types do not need to contain any information. We emit a tentative definition in any compilation unit that references such a **typeinfo** object, and settle for the default initialization of these objects. Such objects will be declared to be `int`, and will have a value of 0 (since this is the value used by the loader to initialize tentative definitions). The only use which can be made of such objects is address comparison. Since the first field of a real **typeinfo** object is "size", a size of 0 indicates a **typeinfo** object for a non-class type, and none of the other fields are present. Note that this discussion does not apply to pointers and references to *classes*.

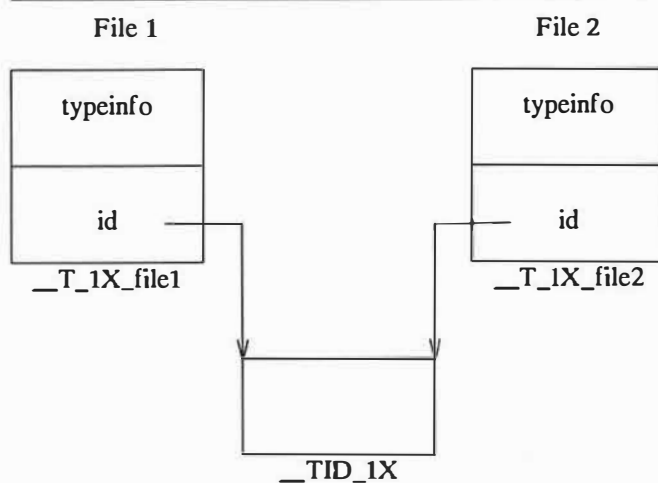
### 3.4 Non-Polymorphic Classes

Non-polymorphic classes, and polymorphic classes which do not have unique virtual tables will not have a unique compilation unit where their **typeinfo** objects can be defined. In such cases, we are forced to emit definitions in *all* files where such **typeinfo** objects are referenced.

#### 3.4.1 Determining **typeinfo** Equivalence

If we emit multiple **typeinfo** objects for the same type, we need a simple way of determining if two **typeinfo** objects represent the same type.

Each **typeinfo** object has a field within it called "id" of type `int*` which is initialized to 0 at compile time for types that have unique **typeinfo** objects. For types which do not have unique **typeinfo** objects, we will initialize the "id" field with the address of a unique *id object* associated with this type. This *id object* is a tentative definition, thus guaranteeing that all **typeinfo** objects for the same type will point to the same *id object*.



**Figure 4.** Multiple `typeinfo` objects for class `X`

---

### 3.5 Determining Subtype Relationships

We use a simple algorithm for determining if one class is a base class of another class. This algorithm must be efficient since we typically need to examine multiple catch clauses before finding one that handles the thrown type. In the previous section, we described why it is sometimes necessary to have `typeinfo` objects point to `id` objects. We have extended this idea so that all `typeinfo` objects point to `id` objects; recall that there is one unique `id` object per class, and it is initialized to 0.

When an exception is thrown, we do the following to determine whether a catch clause can handle the thrown type:

- Traverse the `typeinfo` hierarchy of the thrown type, and mark each of the `id` objects associated with this hierarchy.
- For each catch clause, check the `id` object of the associated `typeinfo` object. If it is non-zero, then this catch clause handles the thrown object.
- Before any user code can be invoked, the `typeinfo` hierarchy of the thrown type must be reset to 0.

Since the `id` objects are global resources, we cannot allow another exception to be thrown while a hierarchy is marked; for developers of thread libraries, we provide a routine which determines whether the run-time is in a non-preemptable state.

## 4. Storage Management

Storage for thrown objects is needed for several reasons:

- Between the time an object is thrown and a matching handler is invoked, user code may be executed (for object cleanup and for the destruction of thrown objects in handlers which are exited). This code could modify the thrown object, but the original exception object must be used to initialize the handler. Hence thrown objects must generally be copied immediately into storage protected by run-time support.
- When a rethrow occurs, it is necessary to have the value and the type of the most recently thrown object.

Storage for thrown objects usually follows a stack discipline, but this assumption can be violated. A thrown object must be retained until its associated handler has exited, although a new exception may

be thrown while in a handler and caught outside it. Figure 5 illustrates this case. The throw of "whatever" causes storage for a new thrown object to be allocated. As the handler is "exited" by the throw routine, the previous thrown object (a "something") is deallocated.

---

```
void func() {  
    try {  
        ...  
    }  
    catch(something& s){  
        throw(whatever);  
    }  
}
```

---

**Figure 5.** Storage for Thrown Objects is not LIFO

We handle this complication by pretending we are managing a stack of thrown objects, but we have two deallocation operations: A *combine* operation coalesces the topmost entry with the one immediately beneath (producing a larger topmost entry). The storage is not actually deallocated. Combine is invoked when the throw routine discovers that a handler in execution will be exited because of an exception. A *deallocation* operation causes storage for the topmost entry to be reclaimed. It is invoked when a handler terminates.

The storage manager begins with a block of static storage which should be sufficient for many programs. This increases the likelihood that exceptions can be thrown even if the heap is corrupted or exhausted. If additional storage is needed, by default the global `new` operator is invoked to obtain additional space. A program can request that a different memory allocator be used by first calling the function `set_eh_new()` with a function pointer. The pointed-to function will then be used to allocate additional storage. Space obtained dynamically in this way is not freed; instead the storage manager's deallocation operation adds unused storage to a free list.

## 5. Object Cleanup

The main goal in the design of object cleanup is to minimize the incremental execution cost for code that does not throw exceptions. Objects in the following categories are subject to cleanup:

1. Partially and fully constructed auto objects which will be out of scope at the catch point. We use the term *partially constructed* to refer to both partially constructed and partially destroyed objects, since construction and destruction order is symmetric.
2. Partially constructed dynamically allocated objects. In addition to cleaning up the object, memory allocated for the object must be deallocated.
3. Partially constructed static objects that have function scope.
4. Partially or fully constructed static objects must be destroyed when `exit` is called.

Objects that do not belong to these four categories do not require cleanup actions. We call objects in these four categories "cleanup subjects".

We use the following terminology in the rest of this document:

### Cleanup Region

A *cleanup region* surrounds a segment of code such that the *destructor counter* value can be used to determine the sequence of cleanup actions necessary for the region. Cleanup regions are made as large as possible to minimize the run-time cost for switching regions. Due to the symmetry of the construction and destruction sequence for auto objects within a block, an entire block can always be a single cleanup region. For dynamic objects and local static objects, a cleanup region will surround the call to the constructor or destructor. Cleanup regions can be



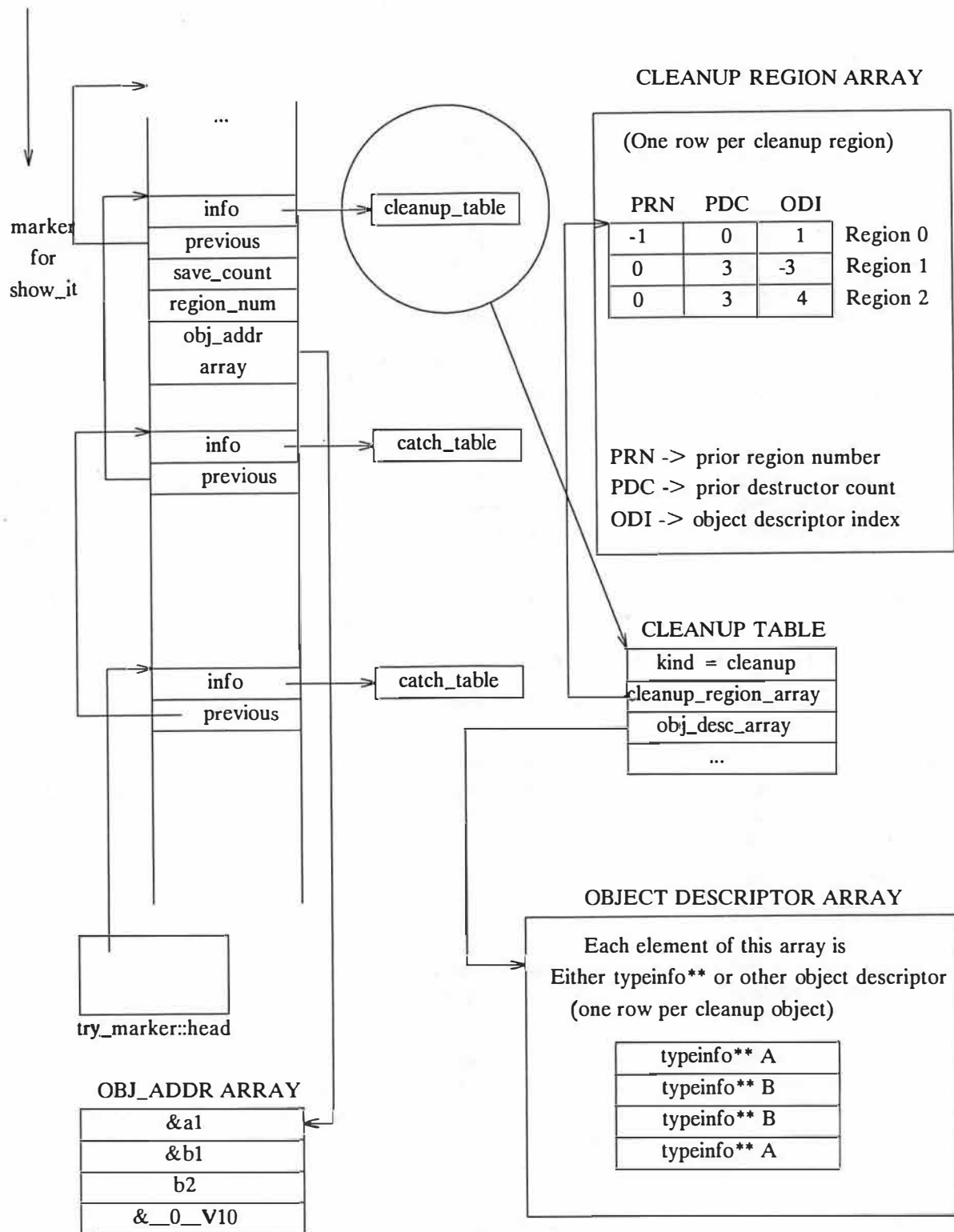


Figure 6. Data Structures for Object Cleanup

nested for various reasons, such as conditional control flow and inlined functions.

#### Destructor Counter

A *destructor counter* is a global variable that represents the absolute number of sub-objects that have been constructed and are subject to cleanup actions in the event of a throw. For example, if class C inherits from class B and there is an object "cobj" defined to be of type C, a destructor count of 2 indicates that "cobj" is fully constructed; a value of 1 indicates that "cobj" is partially constructed, with only sub-object B being fully constructed. The *destructor counter* is often referred to in relative terms. The construction state of an object is expressed as a *destructor counter* value relative to the construction state of its sub-objects; the construction state of a group of auto objects is expressed as a *destructor counter* value relative to the beginning of the cleanup region; the construction state of the beginning of the region is expressed as a *destructor counter* value relative to the beginning of the function. Modifications of the global destructor counter are usually performed by constructors and destructors.

The object cleanup implementation uses two types of data structures:

- Static cleanup tables are generated at compile time. There is one such table for each function requiring object cleanup. These tables describe cleanup regions and objects within cleanup regions.
- Cleanup markers are initialized at run-time and are chained into the same list as **try** and specification markers; they always have a pointer to the corresponding cleanup tables. Cleanup markers are generated for every function that requires cleanup actions and are searched in the reverse order of creation.

A pictorial representation of the cleanup data structures is shown in figure 6. A program example that provides details of the work done during cleanup can be found in the next section.

### 5.1 Program Example for Object Cleanup

The program fragment in figure 7 contains cleanup subjects of types A and B. The static data structures for "show\_it" are depicted in figure 6.

The entry sequence of function "show\_it" is depicted in figure 8. The entry sequence consists of chaining the cleanup marker, saving the *destructor counter*, and setting pointers to the cleanup tables and the vector of addresses for the cleanup subjects. The exit sequence of "show\_it" is also depicted in figure 8; the cleanup marker is unchained.

A listing of the pseudo C code for the examples is shown in figure 9.

- Example 1  
A throw is executed in the routine `illustrate_init`. At this point, none of the objects of "show\_it" have been constructed. The cleanup information has been properly initialized and no cleanup actions are taken as execution resumes at the catch point.
- Example 2  
A throw is executed in the routine `bitwiseB` (note: object b1 is constructed by a bitwise copy; hence we increment the *destructor counter* by 2 after the assignment). The relative *destructor counter* value for "show\_it" is 1, since an A object has been constructed. From this *destructor counter* value, the *cleanup region* value (zero), and the **typeinfo** for the type of A (found in the object descriptor vector), the run-time determines that cleanup subject a1 should be destroyed.
- Example 3  
A throw is executed in the constructor for B. The current *cleanup region* is 1. The relative *destructor counter* value for function "show\_it" is 4 (1 for a1, 2 for b1, plus 1 for sub-object C of b2) and the *destructor counter* value for the start of region 1 is 3 (from diagram 6), yielding a relative *destructor counter* value of 1 for the region. Given this information and the **typeinfo** described in the object descriptor information for the region, the run-time determines that object b2 is partially constructed and destroys the "C" portion of object b2. Additionally, the object

---

```

//Class A has no base classes, and defines a copy constructor
//Class B has one base class, and has no copy constructor
void show_it () {
    int a = illustrate_init();           // Example 1: throw in body of illustrate_init
    A a1;
    B b1 = bitwiseB();                   // Example 2: throw in body of bitwiseB

    if (example == 3) {                  // Example 3
        B::throw_in_constructor = 1;     // throw in constructor B after
        B* b2 = new B;                   // sub-object C has been constructed.
    }
    else {
        if (example == 4 || example == 5) { // Example 4
            illustrate_copy(static_A);     // throw in routine illustrate_copy
        }
    }
    if (example == 5)                     // Example 5: Do a throw to
        throw 5;                          // demonstrate setting of
}                                           // region numbers

```

---

**Figure 7.** C++ Code Fragment for Cleanup Examples

---

```

// Entry sequence for the function "show_it"

clnp_marker.marker::previous = marker::head;
marker::head = (struct marker *) & clnp_marker;
clnp_marker.marker::info = (void *)& cleanup_static_info_obj;
clnp_marker.cleanup_marker::save_count = dt_count;
clnp_marker.cleanup_marker::obj_addr = (void **)obj_addr;

// Exit sequence for the function "show_it"

marker::head = clnp_marker.marker::previous;

```

---

**Figure 8.** Entry and Exit Code for show\_it

descriptor index is negative, indicating that the object was allocated with **new**, and the run-time deallocates the memory for b2. Processing continues at the parent *cleanup region* 0. The *dt\_count* is now 3 (decremented by 1 in `~C()`), and from that value and the **typeinfo** found in the object descriptor array, the run-time determines that destructors need to be invoked on the fully constructed objects b1 and a1 respectively.

- **Example 4**  
A throw is executed in the routine "illustrate\_copy". The *destructor counter* value for the function is 4 and the *cleanup region* is 2. The relative *destructor counter* value for region 2 is calculated to be 1 and the temporary `__V10` (created because of the copy construction in passing the arguments to "illustrate\_copy") is destroyed. Processing continues at parent *cleanup region* 0 and b1 and a1 are destroyed.
- **Example 5**  
This follows the same execution path as example 4; however, no throw is performed in "illustrate\_copy" and object `__V10` is destroyed at the end of the block. The relative *destructor counter* value is calculated to be 3 and the *cleanup region* number is 2. Since the minimum *destructor counter* value for region 2 is 3, no cleanup actions are required. Processing continues

---

```

// Start of region 0
clnp_marker.cleanup_marker::region_num = 0;
a = illustrate_init();
obj_addr[0] = (void *)& a1, A::A(&a1);
obj_addr[1] = (void *)& b1, (b1 = bitwiseB()), dt_count += 2;

if (example == 3 ) {
    B::throw_in_constructor = 1;
    struct B *b2, *__B9;

    b2 = ((__B9 = (struct B *) new (sizeof(struct B))) &&
        (clnp_marker.cleanup_marker::region_num = 1, //start of region 1
        obj_addr[2] = (void *) __B9 ,
        B::B(__B9),
        dt_count -= 2,
        clnp_marker.cleanup_marker::region_num = 0) // reset to prior
        __B9; // region
    }
else
    if (example == 4 || example == 5) {
        struct A __V10 ;
        clnp_marker.cleanup_marker::region_num = 2; // start of region 2
        illustrate_copy
        ((obj_addr[3] = (void *)& __V10, // copy constructor to
        A::A(const A&) (& __V10, &static_A)); // pass argument

        A::~~A(& __V10, 2); // destructor sequence
    } // for region 2

if (example == 5 ) {
    struct thrown_object * __E11; // throw an exception
    __E11 = thrown_object::allocate (typeidB, 4, 0);
    *((int *) __E11->thrown_object::buffer) = 5;
    thrown_object::do_throw(__E11);
}

B::~~B(& b1, 2); // destructor sequence
A::~~A(& a1, 2); // for region 0

```

---

Figure 9. Pseudo C Source for C++ Code in Figure 7

at *cleanup region 0* and *b1* and *a1* are destroyed.

## 5.2 Object Descriptors

In the program fragment shown in figure 7, an *object descriptor* is the `typeid` of the object. However, the information available in the `typeid` is not always sufficient to describe the cleanup actions for an object (e.g., for array objects). For such cases, the *object descriptor* will point to another data structure. The first word of this data structure is 0 to distinguish it from a pointer to `typeid`. The second word identifies the specific kind of object descriptor. The rest of the object descriptor will provide additional information about cleanup actions required.

### 1. Vector Sizes

This is the number of elements for objects that are vectors. For sizes that are not statically determinable, the vector size will be stored by the run-time during allocation of the object.

## 2. Non-Deallocation of Memory

For partially constructed objects that are being created using placement **new** or are being destroyed explicitly, memory for the object should not be deallocated.

## 3. Global **new** and **delete**

The default deallocation for an object is performed using the **delete** operator specified in the **typeinfo** for the class. If the explicit scope operator is used on a **new** or **delete** operation, the object descriptor allows the run-time to determine this.

## 4. Virtual Destructors

If a partially constructed object occurs as a result of a virtual destructor call, this object descriptor indicates where the virtual table pointer is within the object. A virtual table contains a **typeinfo** pointer which is used by the run-time to destroy the object.

## 5. Thrown Object Copy

During a **throw** operation, a copy constructor may be invoked to make a copy of the thrown object to a run-time buffer. If the copy constructor throws an exception, then this object descriptor informs the run-time that a buffer contains a partially constructed object which should be destroyed.

# 6. Exception Specifications

Exception specifications are implemented using elements derived from other aspects of our implementation. On entry to a function with a specification, a *specification marker* is pushed onto the global marker list. The marker points to a static structure which describes the types the function may throw. This structure is identical in form to the one used to represent a list of catch clauses.

When the run-time routine for throw encounters an exception specification, it checks the thrown type against the specification list, applying the same algorithm used to match a thrown type with a set of catch clauses. If the exception does not match any type in the specification list, the routine **unexpected()** is called.

# 7. Performance Analysis

The following analysis expresses the cost of exception handling in terms of the extra C code emitted, as well as the size of static tables produced. This analysis shows the costs per instance of a particular exception feature. Overall costs are dependent on coding style. Refer to the section entitled "Performance Measurements" for some empirical measurements of exception handling costs.

## 7.1 Object Cleanup Costs

Costs are affected by the number of objects in the program, inlining, and other factors.

### 7.1.1 Run-time Costs

1. Five assignments upon entry into a function that includes cleanup information. One assignment at the end of the function to unchain the marker. (Note: for a function that is inlined and has cleanup subjects, cleanup information is incorporated into the containing non-inlined function. Hence an inlined function does not have a marker chaining cost.)
2. Setting the cleanup region number.
  - Two assignments for dynamic objects and function static objects: one to assign the current region number, and a second to reset the previous region upon completion.
  - One assignment on entry into a block containing destructible auto objects.
  - One assignment to initialize the region number on entry into a function with more than one cleanup region.

3. Incrementing the destructor counter.
  - The destructor counter is incremented by 1 in every constructor (including copy constructors) of a class with a destructor.
  - The destructor counter is incremented by the destructor count of a class when an object is copied using bitwise copy.
  - The destructor counter is decremented by 1 in the destructor.
4. One assignment is required to store the address of each cleanup subject.

### **7.1.2 Static Storage Costs**

1. Seven words for the cleanup table (per function with cleanup subjects).
2. Three words for every cleanup region.
3. One word for an object descriptor for each cleanup subject. Three or four words for each special object descriptor.

### **7.1.3 Stack Storage Costs**

1. Six words for the cleanup marker (per function with cleanup subjects).
2. One word for every cleanup subject's address.

## **7.2 Cost of Try Block Entry**

### **7.2.1 Run-time Costs**

1. Two assignments for chaining and one assignment for unchaining the try marker.
2. One assignment to initialize the state of the try marker.
3. One assignment to initialize a pointer to the static information.
4. One call to a run-time library routine if deallocation of a thrown object is needed (i.e., execution exited a catch block). The call is surrounded by the saving and restoring of the destructor counter.
5. The cost of "setjmp" to store the registers.

### **7.2.2 Storage Requirements**

1. Stack storage is required for a setjmp buffer, plus 24 additional bytes for a try marker.
2. 20 bytes for the catch table.
3. 8 bytes per catch clause entry.

## **7.3 Cost of Specifications**

### **7.3.1 Run-time Costs**

1. Two assignments to chain and one assignment to unchain the specification marker (per function with a throw specification).
2. One assignment to initialize a pointer to the static information.

### **7.3.2 Storage Costs**

1. 8 bytes of stack storage per specification marker.
2. 12 bytes per specification table.
3. 8 bytes per item in the specification list.

## 7.4 Cost of `typeinfo`

### 7.4.1 Storage Costs

1. Non-class types
  - 4 bytes for every unique type.
2. Classes
  - 36 bytes for the `typeinfo` of the class.
  - 12 bytes for every base class.
  - 12 bytes for every destructable member.

### 7.4.2 Duplication of `typeinfo`

1. One unique instance for every non-class type.
2. One unique instance for every class that has at least one virtual member function which is not inline.
3. One instance per compilation unit for class types that don't satisfy the above condition, when the compilation unit
  - has cleanup subjects of the type.
  - uses the type in a specification list.
  - has a handler for the type.
  - throws the type.

## 8. Non-Portable Compiler Implementation

Expressing exception handling constructs using portable C causes certain limitations upon an implementation: no assumptions can be made about the layout of stack frames and code addresses cannot be made available to the run-time. A non-portable implementation does not have such restrictions. Combining the elimination of these restrictions with a low-level unwind facility, significant improvements were made in the following areas:

#### Elimination of the marker chain

The compiler can communicate the frame offset of cleanup, try, and specification markers to the run-time through static tables. As the stack is unwound, the run-time determines which frames have markers. Hence there is no cost for chaining and unchaining of markers.

#### Zero cost entry into try blocks

The low-level unwind facility can provide sufficient context to resume execution at the beginning of a catch clause (aided by static tables that contain code addresses corresponding to the beginning of catch blocks). Thus a `setjmp` at the start of a try block is no longer necessary. However, the cost of performing a throw is increased. This cost is proportional to the number of activations that must be unwound, and the number of functions in the application.

#### Statically known object addresses

Instead of using auto object addresses, the run-time can access objects through statically generated tables of offsets within a stack frame. Space cost is reduced for recursive functions, since there is only one instance of the static table. Moreover, the application does not incur the cost of assignments to the object address array.

An additional improvement that could have been made was a zero execution time cost to establish the context for cleanup. Given a program counter value and static tables that describe code ranges within a function, it is possible to provide the same logical information that the portable implementation provides using the *destructor counter* and *region numbers*. We required our portable and non-portable implementations to be compatible; thus we were unable to pursue this zero cost

scheme.

Although the execution cost of such a scheme would be zero, the static storage cost will be higher. The static tables would contain an entry for each code region in which a sub-object was constructed or destroyed. In addition, if an optimizer chooses to reposition code which was described by a table entry, multiple entries may be needed for the same logical section of code.

## 8.1 Optimization in the Non-Portable Implementation

Implicit paths exist from any call in a try block to the set of catch clauses. We make these paths explicit by inserting dummy conditional branches from each call in a try block to the start of the catch clauses. These branches are removed after most optimization phases have run. Care must be taken to prevent the optimizer from inserting code between a call and the following dummy branch, as this code will not be executed if the called routine throws an exception.

## 9. Performance Measurements

### 9.1 Cost of Object Cleanup

Two applications were chosen to measure the costs of object cleanup; the *cfront* translator and the *iclass* utility that is supplied with the *InterViews* [5] package. Neither of these applications throw exceptions or contain try blocks. Therefore, the only cost being measured is the bookkeeping being done to implement object cleanup. Three factors were measured: execution speed, code expansion, and storage requirements. All items are expressed as the percentage difference between an application built with and without exception handling.

The storage requirements of an application compiled with exception handling are dependent on the code size of the application, rather than the data size of the application. A larger application will generally have more classes (resulting in more `TypeInfo` information), and more cleanup subjects (resulting in more cleanup tables). We therefore express the data size increase as a percentage of the original code size, rather than as a percentage of the original data size.

We built a *cfront* which contained cleanup code and cleanup tables (we'll call this an *EH cfront*). We measured the time it took an *EH cfront* to compile a standard application; the execution time increased by 8 to 9 percent. The size of the *EH cfront* code space was 16 percent larger than the *non-EH cfront* code space; the size of the data space increased by 64K (9%). For the non-portable implementation, the performance degradation was 1 to 2%, code expansion was 9%, and data expansion was 77K (10%).

For the second application, *iclass*, we measured the time spent parsing a large set of header files and displaying the initial screen. For the portable implementation, no performance degradation was measurable, the code expansion was 10%, and the data expansion was 34 (9%). For the non-portable implementation, the corresponding numbers were 0%, 6%, and 45K (12%) respectively.

*cfront* experienced a significant performance degradation. Part of this degradation is attributable to the fact that this application constructs and destroys a large number of objects. No performance degradation was discernible for *iclass* since the time spent in actual *InterViews* code is minimal compared to the time spent in the X libraries.

We need to further analyze the data storage costs to determine the distribution of storage costs between `TypeInfo` and cleanup tables. Although *iclass* and *cfront* use different programming styles, the data expansion was similar between the two applications. *iclass* had 134 classes for which `TypeInfo` was required, while *cfront* required 73 `TypeInfo` objects. This is counterbalanced by the fact that *cfront* does not use virtual functions. *iclass* does use virtual functions, which allow the generation of unique `TypeInfo` for most classes.

### 9.2 Cost of try Blocks and throw Execution

A short program that executed a for loop 10,000 times was written. Each iteration of the loop would call a function that would in turn call another function and so on, until 10 frames were on the stack.



There were three versions of this program: one as described, one that surrounded the first call with a **try** block (illustrating the cost of entering a **try** block), and one which did a **throw** from the tenth frame. No objects requiring cleanup were used.

	Plain loop	Try block in loop	Try block and throw in loop
Portable	5	16	75
Non-Portable	5	5	1800

As discussed earlier, entry into a **try** block has zero cost for the non-portable implementation in terms of execution time. However, when a **throw** takes place, the time spent in the run-time unwinding the stack to the catch point is considerably longer than the time it takes to perform a "longjmp" in the portable implementation.

## 10. Language Discussion

Early implementations of a new language feature usually tend to uncover areas where the language definition is not sufficiently precise. This section describes some of the language issues which arose during our implementation effort.

- Destruction of Partially Constructed Heap Objects

When an exception occurs, all automatic objects between the throw point and the handler need to be destroyed. What happens if an exception occurs during the construction of a heap object? There is no mechanism for a programmer to destroy a partially constructed heap object. If a heap object is partially constructed, we treat it like an automatic object; that is, we destroy its fully constructed sub-objects. Furthermore, to minimize memory leaks, we attempt to use the appropriate delete operator to deallocate the memory allocated for the heap object. Memory cannot be deallocated if it was obtained by a placement **new**, since the run-time cannot determine how the programmer planned on deallocating the memory. Perhaps the language could define a mechanism for the programmer to inform the run-time how such memory should be freed.

- Where should the run-time get memory from?

The run-time must make a temporary copy of a thrown object; this temporary copy can only be deallocated upon exit from the handler, since a rethrow operation will reference the temporary copy. Since an exception can occur while inside a handler, the run-time must be prepared to store an arbitrary number of temporary copies of thrown objects. It is essential that the run-time maintain a preallocated pool of memory for this purpose, since the free store may be exhausted or corrupt. However, there is always the possibility that a program will exhaust the preallocated memory. This means an implementation must have a mechanism for acquiring additional memory from the heap if necessary. We chose to introduce a library routine called **set\_eh\_new** to specify the memory allocator to be used; by default, we will use the global **new** operator.

- Cleaning up global objects

The **exit** routine may be called at any time in an application. This routine must be prepared to destroy all static objects which have been constructed; it should also handle any static objects which have been partially constructed.

## 11. Future Work

This section describes various areas where further improvements are possible.

- The tables generated for exception handling take up a significant amount of space. While making measurements for this paper, we observed some instances where the storage cost can be reduced

by using data compression techniques.

- There are various cases where some of the information emitted in the `typeinfo` structures can be omitted. For example, if it can be determined that a particular class cannot possibly result in a partially constructed object, it would not be necessary to emit information about the destructible members in the class.
- A compiler-based implementation can reduce the incremental execution cost to zero; however, this results in an increase in storage costs. Furthermore, the execution cost of a `throw` becomes dramatically higher. Further experimentation with the compiler-based implementation will help us find a reasonable balance between execution and storage costs.
- Inter-procedural analysis can yield a number of improvements, such as the elimination of unnecessary cleanup regions, and a reduction in the number of objects that can potentially be in partially constructed states.
- Duplicate `typeinfo` objects can be eliminated during a pre-link phase. For example, the *cfront* 3.0 uses such a mechanism to eliminate duplicate template instantiations, and this mechanism could be extended to handle `typeinfo` objects.

## 12. Conclusion

We have described our implementation of C++ exception handling. Our portable implementation has made it possible to provide exception handling on three different Hewlett-Packard platforms. On those platforms where support for non-portable techniques exists, we have produced an implementation with minimal performance degradation.

Although we have shown that a portable implementation is possible, the requirement that objects be cleaned up in the event of an exception makes it difficult to produce a portable implementation with acceptable performance. Execution and storage costs associated with exception handling must be paid by all programs which use destructible objects, even if exceptions are not used.

Part of the reason for the rapid proliferation of C++ is the widespread availability of *cfront* on many platforms; any platform that supports a C compiler can support C++. Having the same C++ compiler on multiple platforms ensures a *de facto* language standard while the language is undergoing formal standardization. However, there are some disadvantages to the translator approach:

- The time it takes to compile a source is usually somewhat longer with a translator than with a native compiler.
- Debugging translated code can often be a cumbersome process. However, it has been shown [4] that debugging of translated code can be made transparent.

These two drawbacks are relatively minor. The run-time performance of translated code is roughly equivalent to the performance of directly compiled code. This has been an important factor in ensuring the continued popularity of the translator approach. With the advent of exception handling, we have shown that translated code will have a performance degradation when compared with code produced by native compilers. We believe that portable implementations should attempt to incorporate a few platform specific non-portable techniques in order to provide execution performance comparable to that of code produced by native compilers.

Our native compiler implementation shows that the execution costs can be reduced to acceptable levels, although the storage costs are still significant. Our compiler implementation uses a stack unwinding mechanism which is available in the Hewlett-Packard Precision Architecture run-time environment; such support is not available in all environments.

It is too early to tell whether the costs associated with the C++ exception handling model are worth the benefits of the functionality. As more implementations of C++ exception handling become available, we expect this to be the subject of lively debate within the C++ community.

### 13. Acknowledgements

We would like to thank Margaret Ellis, Jonathan Shopiro, and Glen McCluskey for helping us clarify various languages issues which arose as we proceeded with our implementation. Peggy Chen made significant contributions to the implementation effort.

### 14. References

- [1] Margaret A. Ellis, Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Massachusetts. 1990.
- [2] Bjarne Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Massachusetts. 1991.
- [3] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference Proceedings, April 1990.
- [4] Dmitry Lenkov, Shankar Unni: *C++ Symbolic Debugging*. C++ at Work'90 Conference Proceedings, 1990.
- [5] Mark A. Linton, John M. Vlissides, and Paul R. Calder: *Composing User Interfaces with InterViews*. Computer, 22(2):8-22, February 1989.



# An Assertion Mechanism based on Exceptions

Philippe Gautron  
Université Paris VI, LITP-IBP  
4 place Jussieu, 75252 PARIS CEDEX 05, France  
gautron@rxf.ibp.fr

## Abstract

This paper discusses the concept of assertion in C++ and implementation alternatives for assertion mechanisms. An implementation based on the C++ exception mechanism is described in detail.

We first present a C++ alternative to the `assert` facility provided by the standard C library. It is shown that an assertion mechanism based on specific C++ facilities, such as templates and exceptions, provides more powerful support. The notion of assertion is then discussed both briefly in a formal way and in the particular context of C++.

Experience shows that the use of exceptions must be disciplined. The concept of a *filter class* is introduced to isolate the use of assertions and, as a result, to discipline the use of exceptions.

## 1 Introduction

The goals of this paper are to introduce the concept of assertion in C++ and to present a programming support for the use of assertions through the concept of filter class.

The starting point will be the `assert` macro supplied by the ISO-C library. Use and limitations will be briefly discussed through a C++ version of this macro.

An *assertion* is a predicate stating something that is expected always to be true. Expressing the concept of assertion in a formal way would require an appropriate formal language. But to be useful in a programming language requires assertions to be expressed in a syntax close to the syntax of the programming language.

The `assert` macro is intended to be used with traditional (C and C++) environments. Evolution of technology (distributed and concurrent environments for example) requires more complex error recovery mechanisms and a better user control over these mechanisms. This is why an exception handling mechanism has been recently introduced in C++ [Ellis and Stroustrup 90, Stroustrup 91b]. It is then tempting to associate assertions and exceptions. On the other side, experience drawn from the use of similar exception mechanisms shows that the use of exceptions must be disciplined. Our concept of a *filter class*, based on assertions, is intended to be programming support for such a discipline.

The paper is organized as follows. First, a C++ version of the C `assert` macro is presented. Then, we describe a model of assertion based both on exceptions and templates, we discuss the

notion of assertion both briefly in a formal way and in the particular context of C++, and we introduce the concept of filter class as a concept permitting isolation of the use of assertions. Finally, we present variants in the use of filter classes and we compare our assertion mechanism to different alternatives.

## 2 The C assert Macro

[ISO-C 90] defines `assert` as a macro indicating that its expression argument is expected to evaluate to true at the point of its call. If the assertion is false, a diagnostic comment is written on the standard error output and a call to the `abort` primitive is performed. The `assert` macro refers to another macro, `NDEBUG`, which *must* be set to indicate that debugging is disabled.

The `assert` macro is intended to irrevocably terminate a program after detection of a program failure. A trivial example is null pointer testing, as in:

```
class String {
    // ...
public:
    String (const char* p){
        assert (p != 0);    // aborts the program if p is null
        ...
    }
};
```

Validity of the expression is checked and a rudimentary message is printed in case of failure. The macro is intended to be used with traditional debuggers, that is, debuggers supporting post-mortem debugging.

Introduction in C++ of C libraries and their support are issues continually addressed by the ANSI/ISO C++ Committee.<sup>1</sup> Particularly, [Clamage 90] presents the `assert` C macro as a special case for the following reasons:

- the `NDEBUG` macro cannot be defined as a constant. The value of this macro may be defined at compile-time: compiling with the option `-DNDEBUG` effectively *deletes* the assertions from the program.
- its definition cannot be turned into an inline function. The diagnostic reported when the assertion turns out to be false includes the “stringification” of the assertion, the line number, and the file name at the check point of the assertion. Since macro substitution occurs *before* inlining, this information is expanded with the corresponding values at the declaration point within the inline function. In the absence of C++ run-time support for this information, the use of macros is unavoidable.
- ISO-C specifies that *no* assertion evaluation occurs when assertions are disabled. A macro support is more definitive than relying on compiler optimization (such as null expressions in control statements).

On the other hand, it is demonstrated in [Koenig 89, p. 82-83] that an implementation of the `assert` macro that should be reliable is not as reliable as expected at first sight. A C++ implementation might look like:

---

<sup>1</sup>The ISO-C library has been recently introduced as part of the C++ Standard, with a list of minor changes.

```

// -- file assert.h --

# ifndef NDEBUG

# include <iostream.h>

extern "C" void abort();

# define assert(expr)    ((expr) || \
    (cerr << "Assertion failed: " << #expr \
    << ", file " << __FILE__ \
    << ", line " << __LINE__ << '\n', \
    abort (), 0))
# else /* !NDEBUG */

# define assert(expr)    (0)

# endif /* NDEBUG */

```

A 0 ends the macro definition after the call to `abort`. This trailing expression prevents the code from presenting a void operand to the `||` operator.

Use of `assert` guarantees that, in case of failure, the exact image of the memory will be dumped. The diagnostic is rudimentary and rather rough. Use of this macro supposes that the environment requires no cleanup to be left in an acceptable state. And [Stroustrup 91b, p. 296] is a discussion demonstrating that error handling supported by an exception mechanism compares favorably with more traditional techniques.

### 3 Our Assertion Mechanism

This section presents an assertion mechanism based on the C++ exception mechanism. An example of its use is then shown.

#### 3.1 The Proposal

The overall goal of our implementation is twofold:

- to provide a default behavior similar to the C `assert` facility.
- to exploit specific C++ facilities, templates and exceptions, to provide a more generic and powerful support than a simple macro.

What follows is the proposed implementation:

```

// -- file assert.h --

# ifndef __ASSERT_H
# define __ASSERT_H

# ifndef NDEBUG

# include <iostream.h>
extern "C" void abort();

```

```

// -- assertion testing
template <class E>
class __assert {

public:
    __assert (int expr, const char *exp, const char* file, unsigned int line){
        if (! expr) throw E(exp, file, line);
    }

    __assert (const void *ptr, const char *exp, const char* file,
                                   unsigned int line){
        if (! ptr) throw E(exp, file, line);
    }
};

// -- specific C++ macro
#define eassert(expr, excep)\
    (__assert<excep> (expr, #expr, __FILE__, __LINE__))

// -- standard exception
class Bad_assertion {
public:
    Bad_assertion (const char *exp, const char* file, unsigned int line){
        cerr << "Assertion failed: " << exp << ", file " << file
            << ", line " << line << '\n';
        abort();
    }
};

// -- C-like macro
#define assert(expr)      (eassert(expr, Bad_assertion))

#define else /* !NDEBUG */

#define eassert(expr, excep) (0)
#define assert(expr)        (0)

#define endif /* NDEBUG */
#define endif /* __ASSERT_H */

```

## 3.2 Discussion and Example

### 3.2.1 The \_\_assert Class

The `__assert` class template defines two constructors that differ in their first argument. The three other arguments are identical and correspond to the information needed for the diagnostic in case of failure of the assertion.

The call of one or the other constructor depends on the type of the expression argument of the assertion. If the expression is a pointer, the argument is converted into a pointer of type `const void*`. If the expression is an arbitrary expression, the expression is converted by the compiler into an integral expression and the argument is of type `int`. For example :

```
void f (const char* p){
```



```

    assert (p);          // p is converted into a const void*
    assert (p != 0);      // (p != 0) is converted into an integral expression
    ...
}

```

### 3.2.2 The assert Macro

The `assert` macro defined above is quite similar to the macro described §2. The sole difference is, in the exception-based version, the creation of an additional object to handle the `Bad_assertion` exception.

### 3.2.3 The eassert Macro

The primary goal of the `eassert` macro is to delegate to the caller responsibility for handling the failure (appropriate error messages, call to `exit` instead of `abort`, ...). A typical example is range checking of a subscript operator, as in:

```

class Vector {                                // a vector of integers
    int *vec;
    int sz;

public:
    int size () { return sz; }
    int& operator[] (int index);              // no bounds check
    // ...
};

class Vector_RC : public Vector {              // range-checking version
public:
    class OutOfRange {                        // exception class
        unsigned int l;
    public:
        OutOfRange (const char*, const char*, unsigned int line) : l(line) {}
        unsigned int line () { return l; }
    };

    int& operator [] (int index);              // checks its argument
};

int& Vector_RC::operator [] (int index){
    eassert (index >= 0 && index < size(), Vector_RC::OutOfRange);
    return Vector::operator [] (index);
}

int get_element (Vector_RC& v, int index){     // arbitrary function
    int element;

    try {                                     // checks index
        element = v[index];
    }
    catch (Vector_RC::OutOfRange& e) {         // exception handler
        cerr << "Vector_RC: range checking failed: index=" << index
            << ", size= " << v.size()
            << ", line= " << e.line()

```

```

        << '\n';
        exit (-1);
    }

    return element;
}

```

Since the exception is thrown before the program failure occur, the environment is not corrupted when the run-time flow returns to the caller.

If an exception is not caught (as could be `OutOfRange`), a call to the standard function `terminate` is automatically performed. The default behavior of `terminate` is to call `abort`. An uncaught exception resulting from a call to `eassert` will thus unwind the stack, unlike a call to `assert`. Calls of local destructors will be performed: such cleanup is sometimes referred to as resource finalization.<sup>2</sup>

However, stack unwinding can alter the conditions under which the failure occurred. The difference between the macros `assert` and `eassert` reflects on the programming support they endorse: `assert` is intended to be used by the end-user (and his program assumed to be a “self-contained” application) whereas `eassert` is intended to be used by library classes (to transfer error recovery to the user). A call to `assert` made by a library class could leave the environment in an undesirable state.

## 4 The Notion of Assertion

This section discusses the notion of assertion. The reader can refer to [Meyer 88, p. 111-164] for a good survey of issues relating to the introduction of the notion of assertion into programming languages.

### 4.1 What Assertions Are

Axioms expressed on (abstract) data types correspond to the following kinds of assertions:

- **preconditions:** the properties that must be satisfied whenever an operation is called. These properties refer to the state of the invoked object and to the operation arguments. No code inside the operation is ever executed (1) before a precondition and (2) after a precondition if the condition is not satisfied.
- **postconditions:** the properties that are guaranteed by an operation when it returns. No code inside the operation is ever executed after the postcondition.
- **invariants:** properties that must be satisfied in a stable state by any object instantiated from the type. Expressing invariants transcends the type operations: they are expressed in terms of operations and do not address the specific requirements of an operation.

### 4.2 Introducing Assertions in Programming Languages

Expressing formal assertions would require a full-fledged formal language that would parallel the programming language. Such an approach is difficult to materialize and cannot be seen as

<sup>2</sup>Cleanup of static objects should require overloading `terminate` in order to use a call to `exit`.

realistic in our context.

An alternative is to state the assertions as rules expressed in a syntax close to the syntax of a given programming language. Semantics can be enforced by a compiler support. This is relatively easy for pre- and post-conditions (they can be expressed as normal statements within a function definition), rather complex for class invariants (they must transcend operation definitions and thus appear at the class declaration level).

With regard to the object-oriented paradigm, a major issue relates to the inheritance mechanism. Assertions fit rather well with the concept of concrete class ([Stroustrup 91b, p. 431]). But a function redefined in a derived class<sup>3</sup> must also inherit and satisfy the assertions defined in its base classes. We have two mechanisms (inheritance and assertion) that interfere each other: [Meyer 88, p. 256] demonstrates that an assertion mechanism can constrain the semantics of inheritance.

Since the C++ concept of inheritance was not designed to integrate assertions, we must rely on programmer discipline to ensure the use and the reliability of the assertions.

### 4.3 Assertions in the Context of C++

Putting the semantics of assertion in the context of C++ addresses different specific issues:

- the mechanism must support the global functions and take into consideration the global environment.
- postconditions and invariants are redundant with function constness. This is not true for preconditions because function arguments can be checked in preconditions.
- compound expressions may be part of a return statement. This is incompatible with the expression of postconditions.
- a (compiler-supplied) copy constructor can be silently called by the compiler after a postcondition (typically when an object is returned by a function). Nevertheless, such a constructor is intended to make object copy and not to modify its state. Moreover, an assignment operator or a copy constructor can be automatically generated by the compiler and, so, cannot be checked. But the nature of these operations make this lack a minor drawback. And in any case, they can be user-defined.
- A precondition on a constructor can only involve its arguments since the object is not yet constructed when the precondition is checked. Note that the invariants should have to be satisfied as postconditions of a constructor.
- A post-condition on a destructor can only involve the global environment since the object is destroyed within the destructor.

## 5 The Concept of Filter Class

Fulfilling object-oriented virtues requires, among others, that class interfaces be simple, that each class address one concern (“one class, one concept”) and that class interfaces be documented. And to be convincing, class implementations must be efficient.

---

<sup>3</sup>This issue concerns only the virtual member functions in C++.

The programming style we would like to encourage is the following:

- to make (concrete) classes as simple and as efficient as possible
- to enforce program validity testing by the use of assertions and to enforce error handling by the use of exceptions
- to isolate assertions (and exceptions) in classes *derived* from (concrete) base classes. We will refer to these derived classes as *filter classes*.

We will introduce the concept of filter class with the support of a typical concrete class, the class `Stack`. The whole example is presented then revised in the following sections.

For the sake of legibility, two macros, `require` and `ensure`, are introduced as mere redefinitions of `eassert`:

```
# define require(expr, excep) eassert(expr, excep)
# define ensure(expr, excep)  eassert(expr, excep)
```

In a same way, a third macro:

```
# define check(expr, excep) eassert(expr, excep)
```

could be defined to check class properties in member function definitions [Stroustrup 91b, p. 418]. We will not use the `check` macro in the rest of this paper.

## 5.1 A Class `Stack`: First Version

First, we introduce the class `Stack_raw` as an efficient and simple implementation of a stack:

```
// -- class Stack_raw, 1st version

template<class T> class Stack_raw {
    T *stack;
    int sz;
    int index;

public:
    Stack_raw (int size) : sz (size), stack (new T[size]) {
        index = 0;
    }

    ~Stack_raw () { delete [] stack; }

    int size () const { return sz; }
    int nb_elements () const { return index; }
    int is_empty () const { return index == 0; }
    int is_full () const { return index == sz; }

    void push (T element){ stack[index++] = element; }
    T pop () { return stack[--index]; }
    T top () { return stack[index-1]; }
};
```

Then we define (1) the class `Stack_filter`, as filter class, to introduce assertions on the member functions of `Stack_raw`, and (2) a macro definition to transparently switch the two classes.

```

// -- class Stack_filter, 1st version

template<class T> class Stack_filter : private Stack_raw<T> {
public:

    // nested exception classes (definitions not provided here)
    class StackFull {
    public:
        StackFull(const char* expr, const char* filename, unsigned int line);
    };

    class StackEmpty {
    public:
        StackEmpty(const char* expr, const char* filename, unsigned int line);
    };

    class StackCorrupted {
    public:
        StackCorrupted(const char* expr, const char* filename, unsigned int line);
    };

    // class interface
    Stack_filter (int nelements) : Stack_raw<T> (nelements) {} // <*** see below (1)

    void push (T element){
        require (is_full() != 0, StackFull);

        Stack_raw<T>::push (element);

        ensure (nb_elements() > 0 && nb_elements() <= size(), StackCorrupted);
    }

    T pop () {
        require (is_empty() != 0, StackEmpty);

        T ret = Stack_raw<T>::pop ();

        ensure (nb_elements() >= 0 && nb_elements() < size(), StackCorrupted);

        return ret;
    }

    T top () {
        require (is_empty() != 0, StackEmpty);

        int old_nelements = nb_elements();
        T ret = Stack_raw<T>::top ();

        ensure (old_nelements == nb_elements(), StackCorrupted);

        return ret;
    }
};

```

```

// -- magic cookie
# ifdef NDEBUG
# define Stack Stack_raw

# else /* ! NDEBUG */
# define Stack Stack_filter

# endif /* NDEBUG */

```

Typical use of the class `Stack` looks like:

```

// -- user example, stack of integers
void push (Stack<int>& stack, int i){
    try {
        stack.push (i);
    }
    catch (Stack<int>::StackFull& x){           // <*** see below (2)
        // ...
    }
    catch (Stack<int>::StackCorrupted& x){     // <*** see below (2)
        // ...
    }
}

```

Nevertheless, this first implementation suffers from two drawbacks:

1. the constructor argument (`nelements`) of `Stack.filter` cannot be checked since it is passed to the base class constructor before entering the body of the derived class constructor.
2. our user example works well as long as the macro `NDEBUG` is *not* enabled. Indeed, when this macro is enabled, our macro definitions will cause the substitution of `Stack<int>::StackFull` for `Stack_raw<int>::StackFull`. An error will be detected since the scope of the exceptions is the class `Stack.filter`.

The following sections will show simple improvements that overcome these surface flaws.

## 5.2 Checking Constructor Arguments

In the constructor for `Stack.filter` above, we could require its argument `nelements` to be strictly greater than zero. First, we must note that a null value is not necessarily an error. Allocate a stack with zero elements could be considered to be legal. Accessing such a stack should be an error otherwise.

In order to check the constructor argument, a heap-allocated instance of `Stack_raw` can be created from a pointer declared in `Stack.filter`. This technique is similar to the technique described in [Stroustrup 91a] and introduced to replace the representation and the operations for an object at run-time. The revised version of `Stack.filter` looks like:

```

// -- class Stack_filter, 2nd version
template<class T> class Stack_filter {
    Stack_raw<T>* stack;                               // introduces an indirection

public:

```

```

// nested exception classes, same as above + NullStack and CrammedStack
// ...

Stack_filter (int nelements) {
    require (nelements > 0, NullStack);      // checks nelements

    stack = new Stack_raw<T> (nelements);    // heap-allocation

    ensure (size() > 0, StackCorrupted);
}

~Stack_filter () {
    require (nb_elements() == 0, CrammedStack);

    delete stack;
}

// to forward the requests
int size () const { return stack->size (); }
int nb_elements () const { return stack->nb_elements (); }
int is_empty () const { return stack->is_empty (); }
int is_full () const { return stack->is_full (); }

void push (T element){
    require (is_full() != 0, StackFull);

    stack->push (element);

    ensure (nb_elements() > 0 && nb_elements() <= size(), StackCorrupted);
}

T pop () {
    require (is_empty() != 0, StackEmpty);

    T ret = stack->pop ();

    ensure (nb_elements() >= 0 && nb_elements() < size(), StackCorrupted);

    return ret;
}

T top () {
    require (is_empty() != 0, StackEmpty);

    int old_nelements = nb_elements();
    T ret = stack-> top ();

    ensure (old_nelements == nb_elements(), StackCorrupted);

    return ret;
}
};

```

This implementation solves the first problem (the constructor argument can be checked) but does not solve the second problem (the scope of the exceptions).

### 5.3 Making Exceptions Visible

Macro substitution defeats our user example §5.1. Different solutions can be proposed:

- introduce conditional macros in the user's code. This is the worst of the solutions.
- declare the exception classes at global scope (with the implied name space pollution).
- transfer the exception classes to the base class `Stack_raw`. This solution works well with our first implementation (`Stack_filter` derived from `Stack_raw`) but fails with the second implementation (when the two classes are not related by inheritance).
- declare the exception classes nested in a global class and use derivation of the surrounding class.

This latter solution lead us to the third implementation, outlined as follows:

```
class Stack_exceptions {
public:
    class StackFull { /* ... */ };
    class StackEmpty { /* ... */ };
    class StackCorrupted { /* ... */ };
    class NullStack { /* ... */ };
    class CrammedStack { /* ... */ };
};

template<class T> class Stack_raw : public Stack_exceptions {
    // ... no change
};

// -- class Stack_filter, 3rd version
template<class T> class Stack_filter : public Stack_exceptions {
public:

    Stack_filter (int nelements) {
        require (nelements == 0, NullStack);
        // ...
        ensure (size() > 0, StackCorrupted);
    }

    ~Stack_filter () {
        require (nb_elements() == 0, CrammedStack);
        // ...
    }

    void push (T element){
        require (is_full() != 0, StackFull);
        // ...
        ensure (nb_elements() > 0 && nb_elements() <= size(), StackCorrupted);
    }
};
```



```

T pop () {
    require (is_empty() != 0, StackEmpty);
    // ...
    ensure (nb_elements() >= 0 && nb_elements() < size(), StackCorrupted);
}

T top () {
    require (is_empty() != 0, StackEmpty);
    int old_nelements = nb_elements();
    // ...
    ensure (old_nelements == nb_elements(), StackCorrupted);
}
// ...
};

#ifdef NDEBUG
#define Stack Stack_raw

#else /* ! NDEBUG */
#define Stack Stack_filter

#endif /* NDEBUG */

```

## 5.4 Moving Inheritance in Nested Classes

In the implementation above, `Stack_raw` and `Stack_filter` are two classes derived from the class `Stack_exceptions`. This relation of inheritance is more driven by an implementation consideration than by a conceptual consideration.

An alternative is to separate the concepts, implementation and exception handling, by using nested classes inheriting from the global exception class. The revised implementation and user code can be outlined as follows:

```

class Stack_exceptions {
    // ... no change
};

template<class T> class Stack_raw {
public:
    class Exception : public Stack_exceptions { };
    // ... no change
};

// -- class Stack_filter, 4th version
template<class T> class Stack_filter {
public:
    class Exception : public Stack_exceptions { };

    Stack_filter (int nelements) {
        require (nelements == 0, Exception::NullStack);
        // ...
        ensure (size() > 0, Exception::StackCorrupted);
    }
};

```

```

    }

    // ...
};

void push (Stack<int>& stack, int i){
    try {
        stack.push (i);
    }
    catch (Stack<int>::Exception::StackFull& x){
        // ...
    }
    catch (Stack<int>::Exception::StackCorrupted& x){
        // ...
    }
}

```

## 5.5 Conclusion

In the different implementations we have presented above, the introduction of exception names in the name space was a major issue. Nested classes are an appropriate support to encapsulate the different exceptions raised by a class, but they convey their own drawbacks:

- useless duplication of classes if the exception classes are nested in a class template: inheritance of a global class encapsulating the exception classes can alleviate this problem.
- sharing of information: nested classes can require the declaration of an intermediate class (`Exception` above), for a purpose of visibility, that does not introduce new functionality.
- overloading of user code: nested classes can require the use of multiple qualified names, such as `Stack<int>::Exception::StackFull`.

However, assertions and filter classes are two concepts that can be adapted to any scheme of exceptions. Introducing exception names in the name space is an issue by itself.

## 6 Variants

The use of the assertion mechanism is not limited to its use for filter classes as described above. This section presents two variants: (1) a user-defined class derived from the implementation class, and (2) a technique to discriminate preconditions from other conditions.

### 6.1 Derived User-defined Classes

Rather than deriving a template filter class from the template base class, we can merely ignore error recovery and delegate to the user full responsibility for dealing with this issue. An example can be outlined as follows:

```

template<class T> class Stack_raw {
    // ... no change
};

// user-defined class

```

```

class Stack_int : public Stack_raw<int> {
public:
    class StackFull { /* ... */ };
    class StackEmpty { /* ... */ };
    class StackCorrupted { /* ... */ };
    class NullStack { /* ... */ };
    class CrammedStack { /* ... */ };

    void push (int element){
        require (is_full() != 0, StackFull);
        Stack_raw<int>::push (element);
        ensure (nb_elements() > 0 && nb_elements() <= size(), StackCorrupted);
    }

    // ...
};

```

The class `Stack_int` above is yet another filter class, and should be a specialized implementation of the class `Stack.filter` introduced §5. There is no real difference (in spirit) between the classes `Stack.filter` and `Stack_int`, except that, in the latter case, all error handling is under the user's responsibility. Whether error recovery must be managed by a class library or by the end-user appears to be a matter of opinion.

## 6.2 Discriminating Preconditions

Setting the `NDEBUG` macro disables any assertion. There is no way to separate preconditions from postconditions and class properties. It can be argued that preconditions must always be checked (whatever the run-time mode is) while postconditions and class properties should be enabled only in debugging mode. Use of macros allows easy selection of the run-time condition. For example:

```

# ifndef NPRECONDITION
# define require(expr, excep) eassert(expr, excep)
# else
# define require(expr, excep) (0)
# endif

# ifndef NDEBUG
# define ensure(expr, excep) eassert(expr, excep)
# define check(expr, excep) eassert(expr, excep)
# else
# define ensure(expr, excep) (0)
# define check(expr, excep) (0)
# endif

class Stack_exceptions {
    // ... no change
};

template<class T> class Stack_raw {
public:
    class Exception : public Stack_exceptions { };
    // ... no change
};

```

```

template<class T> class Stack_filter {
public:
    class Exception : public Stack_exceptions { };
    // ... no change
};

# if NPRECONDITION && NDEBUG
# define Stack Stack_raw
# else
# define Stack Stack_filter
# endif /* NDEBUG */

```

Within this scheme, using `Stack_raw` should require explicit disabling of the preconditions (by defining `NPRECONDITION`) in addition to define the debugging option `NDEBUG`. For example:

```

# use of Stack_raw
% CC -DNDEBUG -DNPRECONDITION ...

# use of preconditions in Stack_filter
% CC -DNDEBUG ...

# use of all assertions in Stack_filter
% CC ...

```

The concepts of assertion and filter class are not altered by this technique and these macros are only syntactic sugar around the concepts.

## 7 Alternatives

This section examines two alternatives to the assertion mechanism presented §3.

### 7.1 A Function Template

[Stroustrup 91b, p. 419] describes an inline *function* template `Assert` that mimics the C `assert` macro:

```

// -- definition of Assert
template <class T, class E>
inline void Assert (T expr, E excep){
    if (! NDEBUG)
        if (! expr) throw excep;
}

// -- example of the use of Assert
class Bad_assertion {
public:
    Bad_assertion ();
};

void f (void *p){
    Assert (p, Bad_assertion()); // <*** see below
}

```

In this example, a `Bad_assertion` exception is thrown if `p` is a null pointer.

This approach suffers from the following drawbacks:

- the object thrown by the exception is created *before* the call to `Assert`. This object will be created, whatever the evaluation of the assertion is. This contradicts our requirement (see §1) that no evaluation must occur when assertions are disabled. More, a call to `abort` *inside* the body of the constructor of `Bad_assertion` will cause a memory dump before testing the assertion.
- stringification of the assertion, line number and file name cannot be directly supported by the `Assert` function (see §1). Introduction of this information might be possible (with contortions) although this would not solve the conceptual issue explained above.

## 7.2 Generalizing Assertion Type

A variation on our first proposal (§3) would be to make the type of the assertion a template parameter. The revised definitions might look like:

```
template <class T, class E>
class __assert {

public:
    __assert (T expr, const char *exp, const char* file, unsigned int line){
        if (! expr) throw E(exp, file, line);
    }
};

# define eassert(expr, excep)\
    (__assert<expr, excep> (#expr, __FILE__, __LINE__))

# define assert(expr)    (eassert(expr, Bad_assertion))
```

The difference between the two approaches is that the former is based on constructor overloading whereas the latter is based on template instantiation: a new class will be created for each pair (assertion type, exception class). It is up to the compiler to create the appropriate instance.

## 8 Related Work

A++ [Cline and Lea 90] is an attempt to extend C++ with a formalism based on assertions. Preconditions, postconditions and class invariants are introduced by specific access specifiers inside the class declaration and stated by rules referring to the class members.

A++ is intended to be used as a front end to a C++ compiler and to support static checking of assertions: the assertions are turned into exceptions only when run-time information is needed. The static analysis allows the compiler to optimize the run-time overhead implied by the assertions and, in that sense, parallels the static type-checking performed by the compiler.

A++ encourages the use of assertions as specifications in abstract base classes. In contrast, the filter classes are primarily intended to be used as classes derived from concrete classes, for testing their validity or as help for debugging.

## 9 Conclusion

In this paper, we have presented a model of assertion that disciplines the use of exceptions. Our assertion mechanism is independent of the exception mechanism, does not rely on a specific C++ implementation, does not require an extension to the language, and is orthogonal to the other language features. The use of assertions, in coordination with the use of exceptions, allow better user error recovery and better failure diagnostics than the traditional macro used in C and C++ environments. More, the assertions allow the user to document *why* the implementation or the execution of a class is correct by stating the tests that are satisfied.

Exception handling has a cost, even when these exceptions are never caught. The implied run-time penalty should have to be paid only if requested by the end-user. The concept of filter class separates class implementation from class testing. An implementer focuses first on the basic functionality supplied by a class and then worries about class testing, without needing to modify the class implementation. Filter classes also provide, with the support of macros, flexibility to select at compile-time the degree of testing supported by the program. The impact of such flexibility on performance can be significant, especially when checked functions, such as access operators with bounds check, are (over)used in a program.

## Acknowledgements

This paper has benefited by discussions with my colleagues from the Library Working Group mailing list of ANSI/ISO-C++. Particular thanks to Tony Hansen for his contributions and to Dennis Shasha and Daniel Edelson for their review of the paper.

## References

- [Clamage 90] Stephen D. Clamage. *ANSI C Library Compatibility Issues*.  
Doc No: ANSI X3J16/90-0105, November 1990.
- [Cline and Lea 90] Marshall P. Cline and Doug Lea. *Using Annotated C++*.  
Proceedings of C++ at Work-90, Secausus (NJ), USA, September 1990.
- [Ellis and Stroustrup 90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Programming Language*.  
Number ISBN 0-201-51459-1. Addison Wesley, 1990.
- [Koenig 89] Andrew Koenig. *C Traps and Pitfalls*.  
Addison-Wesley, ISBN 0-201-17928-8, 1990.
- [ISO-C 90] X3J11/WG14. *Programming Language C*.  
Doc No: ANSI X3J16/90-0013, February 1990.
- [Meyer 88] Bertrand Meyer. *Object-oriented Software Construction*.  
Prentice Hall, ISBN 0-13-629049-3, 1988.
- [Stroustrup 91a] Bjarne Stroustrup. *Sixteen Ways to Stack a Cat*.  
The C++ Report, January 1991.
- [Stroustrup 91b] Bjarne Stroustrup. *The C++ Programming Language*.  
Second Edition. Addison-Wesley, ISBN 0-201-53992-6, 1991.

# A Communication Facility for Distributed Object-Oriented Applications

Salil Deshpande<sup>1</sup>

Pierre Delisle<sup>2</sup>

Afshin G. Daghi<sup>3</sup>

*Sun Microsystems Computer Corporation  
Mountain View, California*

## Abstract

Inter-Object Communication (IOC) is a facility for object-based inter-process communication in distributed object-oriented applications. IOC allows transparent communication between objects — each object can send a message to any other object regardless of the recipient's location. It is currently built on top of the SunOS RPC facility.

In this paper we describe the IOC abstraction and its implementation as a C++ library. IOC provides a design philosophy and tools that completely hide the details of remote communication among objects when developing distributed object-oriented applications. The architecture of IOC is simple and does not rely on low level services of the operating system. Therefore, it can be ported easily to other object-oriented languages and platforms.

## 1. Introduction

Over recent years, the degree of abstraction for inter-process communication (IPC) has increased. Traditional methods using pipes and sockets have given way to Remote Procedure Calls (RPC) [Lyo84] and more recently to more powerful fault-tolerant systems such as ISIS [Bir87]. However, the abstraction most suitable for generic object-oriented distributed applications — transparent and lightweight inter-object communication across address-spaces and machines — is still difficult to find on most popular operating systems, including Unix.

Most approaches to providing transparent communication between remote objects have concentrated on developing complete object-based operating systems and environments. In these systems, objects and object communication are among the base services provided by the operating system, and therefore object-oriented distributed applications are very natural in these environments. Eden [Alm85], Argus [Lis88], V [Che88], and Amoeba [Mul90] are among the well-known examples. Unfortunately, none of these systems are standards in the marketplace, and therefore are generally not used in developing commercial applications.

For the commercial software community, a more useful solution is to develop a complete object-oriented operating environment above the base operating system. Here, the object management facility is completely independent of the hardware and the operating system. This approach is currently being pursued by the Object Management Group (OMG) consortium. Through its Object Management Architecture (OMA), the OMG's goal is to define standards to facilitate object-oriented application development and usage in heterogeneous distributed environments. A key compo-

---

<sup>1</sup>Now with Enterprise Integration Technologies (EIT), Palo Alto, CA 94301. Email: salil@eitech.com.

<sup>2</sup>Email: pierre.delisle@eng.sun.com.

<sup>3</sup>Now with Axil Workstations, A Division of Hyundai Electronics America, San Jose, CA 95134. Email: afshin@axil.com.

ment of the OMA is the Object Request Broker (ORB) [OMG91], which provides the basic object communication and management services. Although a specification for the ORB component has been accepted by the OMG, there is no implementation available as of this writing.

SunSoft currently offers ToolTalk[TT91], which allows for basic inter-application communication and application interoperability on a multi-vendor network, but it is mainly meant to be used for coarse-grain objects. Typically, a ToolTalk object is an entire application. Tooltalk is ideal, therefore, for coarse-grain communication (e.g. integrating various applications on the same desktop) and has been used with success in those areas.

However, our requirements called for a lightweight implementation, support for finer-grained objects, marshalling and unmarshalling of user-defined objects, reasonably fast communication between objects across machine-boundaries, and data caching for performance.

## 2. Communication Model

The principal goal of IOC was to simplify the design and implementation of distributed object-oriented applications by providing software developers with a facility for transparent communication among objects residing in distinct address spaces.

In our model, objects are passive entities. They are identical to the object in the implementation language (C++ in our case). An object exists and is meaningful only in the address-space of a particular process (a local or remote program in execution). Only static interfaces to objects are supported; C++ type-safety is always preserved.

The IOC communication model is an extension of the object and member function invocation paradigm in C++. In C++, the message<sup>1</sup> "doit" is sent to an object of class "Foo" as follows:

```
f = new Foo;
f->doit();
```

In our paradigm, this object of class Foo can exist in some remote address space. If we want to communicate with this object, we must have its *ObjectId*, obtained by earlier transactions. Knowing the *ObjectId* of a remote object, we can invoke a member function of a remote object in the following manner:

```
ObjectId oid;
*
* // oid gets assigned the ObjectId of some remote object
* // of class Foo                                (1)
*
f = new Foo(oid);                                (2)
f->doit();                                        (3)
```

Thus, our communication model requires only three steps for all inter-object communication. (1) We obtain an *ObjectId* of an object. *ObjectIds* can refer to local (in our own address-space) or remote objects. (2) We obtain a *secondary copy* of that object, using an overloaded constructor that takes an *ObjectId* as an argument. A secondary copy of an object is much like the original *primary* copy, in that it contains the same state and supports exactly the same operations as the primary copy. (3) We invoke a member function on the secondary copy. The object 'f' knows that it is a secondary copy, and therefore transparently forwards this message to its primary copy. The member function 'doit' is invoked on the primary copy (in the remote process), and the results are sent back to the secondary copy, which in turn delivers them to the caller.

---

<sup>1</sup>In this paper, *message* and *member function* are used interchangeably to describe operations on objects.



All communication happens in this manner. To communicate with a remote object, we *must* have its `ObjectId`, we *must* obtain a secondary copy of that object, and we *must* invoke the message (member function) on that secondary copy. Our paradigm does not provide any other mechanisms.

The secondary copy is much like a *cache* of the primary copy. Most objects support a 'sync' member function that copies the state of the primary into the secondary. We invoke operations on the secondary copy, and if these operations cannot be performed with cached data, the request is transparently forwarded to the primary copy of that object and the operation is performed on the primary copy in the remote process. Marshalling and unmarshalling of arguments and results also happen transparently.

As the above paragraph implies, there will be some operations invoked on the secondary copy that do not result in communication across processes even if the primary is remote. This is because some requests can be served by the secondary copy itself, with cached data. The details of whether and under which circumstances a particular member function needs to be implemented as a *message* (member function that needs to be forwarded to the primary copy) is part of the object implementation, and is hidden from the client of that object. Therefore, by designing classes properly, such caching of object state can be used to improve performance by the class implementor.

Because objects always stay in the process where they have been created, a client object can always obtain a secondary copy of a remote server object using its `ObjectId`. In fact, several processes can obtain secondary copies of a particular object. The original (primary) object does not have knowledge about the number and location of its secondary copies. When state changes happen in the primary, secondaries are not notified; clients of secondaries must be aware that their respective primaries might undergo changes.

The fact that primaries are independent of their secondary copies, while secondaries rely on the availability of the original object, imposes some restrictions on the implementor. First, objects must be designed to be self contained; i.e., they must not make any assumption about the existence and/or location of their secondary copies. Second, if the primary becomes inaccessible, the secondary copies are crippled. Therefore, the implementor must make sure that objects remain available. This can be done either through replication or through making objects persistent.

### 3. Architecture

We have currently implemented the IOC abstraction as a C++ library. Three classes are defined in the IOC library to implement all the mechanisms required for transparent access to remote objects: `Cobject`, `ObjectId`, and `Liaison`.

A distributed application based on IOC consists of one or more processes running on one or more machines<sup>1</sup>. Any object that wants to be accessed from remote address-spaces must inherit from the abstract class *Cobject*, which stands for *Communicating*<sup>2</sup> object. `Cobject` defines the common structure required by IOC to support communication between remote objects.

The *ObjectId* associated with every `Cobject` unambiguously identifies it in the network. Because our implementation is based on the ONC<sup>3</sup> platform, the `ObjectId` consists of the following four components (see Figure 1): the first two identify the process where the primary copy of the `Cobject` resides (the fully qualified hostname, e.g. `masala.eitech.com`, and RPC program number of

---

<sup>1</sup>Just as in traditional distributed applications based on procedural languages.

<sup>2</sup>Where the term *communicating* implies the support for remote communication.

<sup>3</sup>Open Network Computing platform developed by Sun Microsystems, which includes RPC and the eXternal Data Representation (XDR) routines [Cor91].

the service provided by that process), while the last two identify the Cobject within that process (classId of the Cobject and its sequence number within that class).

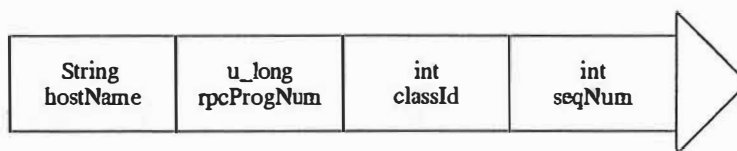


Figure 1: ObjectId

The *Liaison* object, instantiated in every process, acts as the object manager for all primary Cobjects created within that process. Its responsibilities include maintaining a structure that maps ObjectIds onto pointers to the actual Cobjects in that address space, detecting the arrival of pmessages, and dispatching an incoming message to the proper Cobject. Its structure is briefly presented in Figure 2.

```

// Constructors
Liaison(u_long rpcProgNum); // primary -> Cobject(liaisonClassId, true, primary)
Liaison();                 // secondary -> Cobject(liaisonClassId, false, secondary)

// Registration
registerPcobject(int classId, Boolean defaultRcvr, ObjectPtr ptr, String hostName);
void unregisterPcobject(ObjectId objectId);

// Network service information
SVCPRT* transportUdp;
SVCPRT* transportTcp;
u_long rpcProgramNum;

// Pmessages that provide information about objects in my address space
PmessageStatus getClassIdList(intList classIdList);
PmessageStatus howManyObjects(int classId, int& howMany);
PmessageStatus getSeqNumList(int classId, intList& seqNumList);

```

Figure 2: Liaison

Since IOC is based on RPC, the *Liaison* in each address space must listen to the outside world on an RPC program number. The constructor of *Liaison* therefore takes an RPC program number as an argument. If keyword *TRANSIENT* is supplied, a transient RPC program number is allocated to the process. The constructor registers the service with the portmapper under both connectionless (UDP) and connection-oriented (TCP) transport protocols. Being itself a subclass of class *Cobject*, *Liaison* also offers services to provide information about the Cobjects in that address space, such as lists of ObjectIds of a particular class.

As was already pointed out, abstract class *Cobject* (see Figure 3) implements the core facilities to support transparent communication between remote objects.

```

// Constructor used to create a primary copy, or a secondary copy whose ObjectId is
// unknown at creation time
Cobject(int classId, Boolean defaultReceiver=false, CobjectType type=primary);

// Constructors used to create a secondary copy whose ObjectId is known at creation
// time
Cobject(ObjectId objectId);
Cobject(String hostName, u_long rpcProgNum, int classId, int seqNum=0);

// Cobject identification
ObjectId objectId;
Boolean primaryCopy;
setObjectId(ObjectId objectId);
setObjectId(String hostName, u_long rpcProgNum, int seqNum=0);

// Communication parameters (used only for secondary copies)
long timeout;
TransportProtocol protocol; // udp or tcp
int retriesLimit;
setCommParameters(long timeout, TransportProtocol protocol, int retriesLimit);

// Communication functions
Boolean sendMessage(char* rpcArg, xdrproc_t xdrprocArg,
                   char* rpcResult, xdrproc_t xdrprocResult,
                   int rpcResultSize, RpcError& rpcError);
Boolean getRpcArg(SVCXPRT* transport, char* rpcArg, xdrproc_t xdrprocArg, int size);
virtual void receiveMessage(SVCXPRT *transport, int pmessageId) = 0;

```

Figure 3: Cobject

When a primary copy of a Cobject is created, the constructor calls `Liaison::registerPcobject()` to register the object's mapping. Cobject's destructor removes that mapping by calling `Liaison::unregisterPcobject()`. The arguments to the constructor of a secondary copy specify the primary copy referred to. This can be specified through an `ObjectId`, or by specifying each component of the `ObjectId` explicitly. It is often the case that a process only has a single instance of a primary Cobject for a given class. That instance is created with the *defaultReceiver* attribute set and is given the special sequence number 0 so that clients can easily refer to it without having to get its sequence number first. The `ObjectId` of a secondary copy can always be re-specified, so that the same variable can be reused to access different primary copies of the same class.

It is important to note that the group of member functions that actually perform communication operations are only invoked through the communication stubs generated by the `iocgen` protocol compiler described in Section 4.1. As a matter of fact, the code for virtual member function `receiveMessage()` is itself completely generated by the protocol compiler.

Figure 4 illustrates the architecture of a distributed object-oriented application using the IOC library.

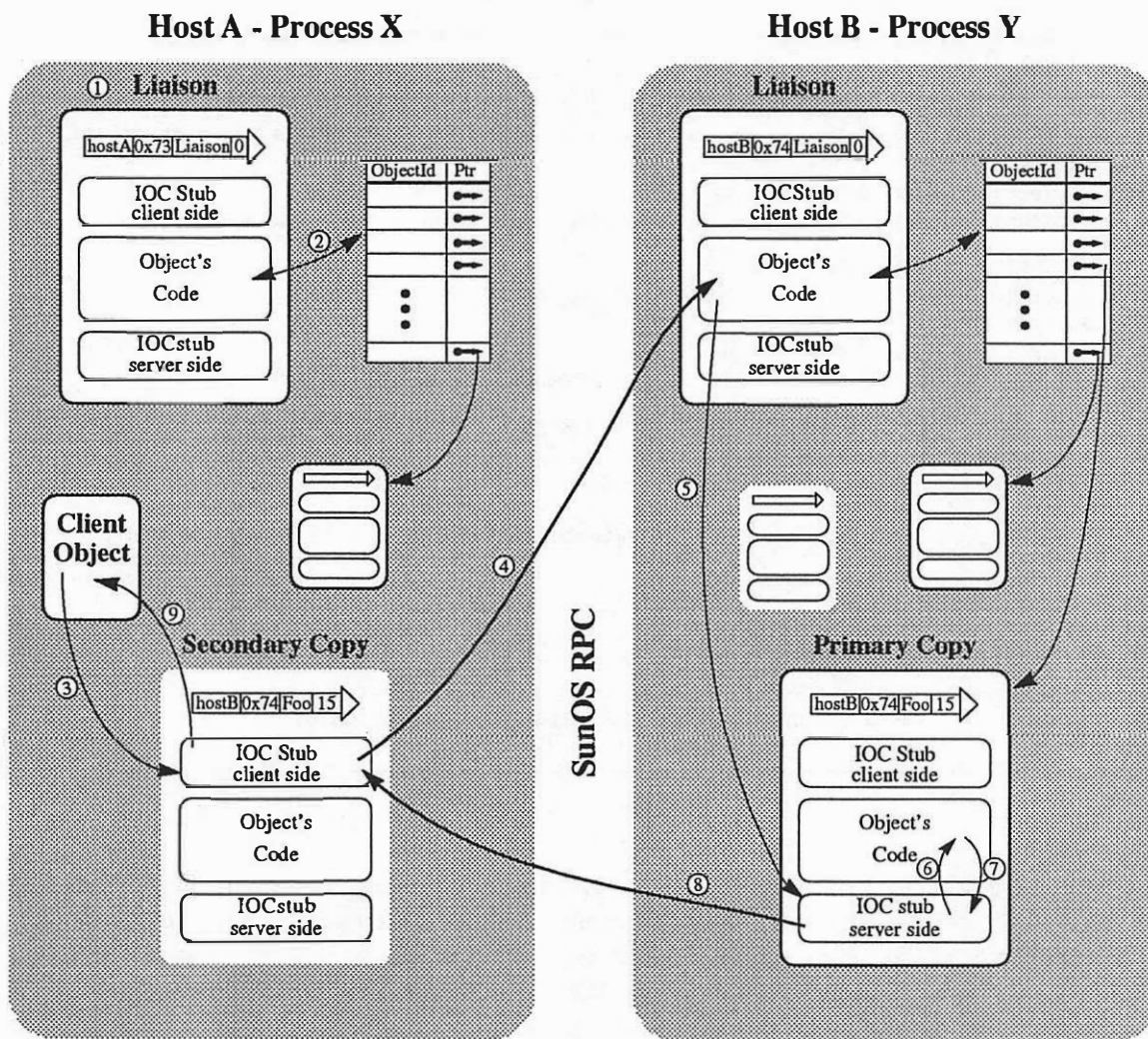


Figure 4: Distributed object-oriented application using IOC

- ① Liaison registers a network service with a specific RPC program number.
- ② Every creation of a primary copy of a Cobject is automatically registered in the object table managed by Liaison.
- ③ A client object sends a message (in this case, a pmessage) to the local secondary copy of the server object.
- ④ The IOC stub marshals the pmessage's header information (ObjectId and pmessageId) and arguments, and makes an RPC call to the process where the primary copy resides.
- ⑤ Liaison receives the RPC request, unmarshals the pmessage's header information, and dispatches the message to the proper object using the object table mappings.
- ⑥ The IOC stub unmarshals the arguments according to the pmessageId, and invokes that pmessage.
- ⑦ The pmessage member function code is executed and returns.
- ⑧ The IOC stub marshals the results and sends them back as a reply to the RPC message.
- ⑨ The IOC stub unmarshals the results and returns them back to the client object.

## 4. Issues in the Design and Use of IOC

### 4.1 Communication Protocol

A crucial issue in the design of a distributed application is the specification of its communication protocol. It should blend well with the general programming paradigm and hide the fact that parts of the application are running on different address spaces, which may reside on different machines. As much as possible, the programmer should not be exposed to the low level communication details. For example, RPC is a nice extension of the normal procedural programming paradigm; a remote procedure is called just like a local procedure.

In the IOC paradigm, Objects act both as clients and servers, depending whether the Object is used as a secondary or a primary copy. The communication protocol is implicitly defined through ObjectIds and the pmessages of the Objects that the application supports. The information provided in a pmessage's signature is however not sufficient to allow proper client-server communication; three important pieces of information are still required.

First, each parameter of a pmessage must be further qualified as being either used as an input argument or as a result. Only input arguments are sent to the server, while only results are returned back to the client. Second, it must be possible for a pmessage to transmit internal state (data members) that is not explicitly specified in the pmessage's signature. For example, a `sync()` pmessage that does not have any parameters could transfer the complete state of a Object from the primary copy to the secondary copy. Third, because clients and servers communicate using a machine independent data representation (SunOS RPC uses the XDR standard), conversion routines must be implemented.

Although it would be desirable to support the definition and implementation of the communication protocol completely within the Object base class, the difficulties encountered while experimenting with this approach led us to privilege the use of a protocol compiler.

The client and server communication stubs are therefore generated through the *iocgen* protocol compiler. A protocol definition file has a ".ioc" extension, and is written using an interface description language (IOCL). In its current status, IOCL is primitive; our primary goal was to have a simple implementation of *iocgen* that would quickly satisfy our needs. It therefore only has 5 keywords (FUNCTION, RPCARGMEMBERS, RPCARGPARAMETERS, RPCRESULTMEMBERS, RPCRESULTPARAMETERS), covering the basic requirements for the definition of a communication protocol. Ideally, the language should be defined with a syntax compliant with CORBA's Interface Definition Language (IDL)[OMG91]. The example below shows how IOCL is used in the definition of one of Liaison's pmessages.

```
FUNCTION PmessageStatus getSeqNumList(int classId, intList& seqL)
RPCARGPARAMETERS int classId
RPCRESULTPARAMETERS intlist& seqL
```

The obvious drawback of IOCL's simplicity is its lack of support for the specification of complex data types. *iocgen* provides built-in support for all fundamental types supported by the XDR routines in the SunOS RPC library<sup>1</sup>, plus a few other ones such as `char**`, `ObjectId`, and `PmessageStatus`. For other types, *iocgen* assumes that a user defined XDR structure is associated with the type, and that implicit type conversions are reciprocally defined for both structures. The actual XDR conversion routine must also be implemented, and *rpcgen* is used to ease this process.

---

<sup>1</sup>*iocgen* produces a .x file later processed by *rpcgen*.

## 4.2 Communication Parameters

Although every effort is made to handle all details of communication with remote objects transparently, there is still a limited set of communication parameters that is available to the developer to allow him flexibility in the implementation of a distributed application.

The communication parameters are defined as data members of the Cobject base class. They take effect when forwarding a pmessage from a secondary copy to its primary copy. They are the transport protocol, the time-out value, and the maximum number of retries for deadlock avoidance.

The invocation of a pmessage can either be synchronous or asynchronous. Asynchronous communication is achieved by setting a null time-out value. Callback capability can easily be achieved by using on the client side a primary Cobject that acts as a callback service. The ObjectId of this callback object simply needs to be known by the recipient of the asynchronous message (could be specified as an argument to the pmessage).

In a synchronous invocation, the pmessage blocks the process (or thread if the application is multi-threaded) until it either succeeds (reply is received), fails, or times out. The value of communication parameters depends on the context in which the pmessage is invoked.

If the client side of the application wants to block when it invokes the pmessage, the time-out value may then be set to a long interval. This is the case when the client side does not have anything else to do besides waiting for the pmessage's reply, or if it is multi-threaded, where the waiting does not prevent other threads of the process to execute normally.

However, we also wanted to support the case where a process contains both secondary and primary copies of Cobjects, and does not have multi-threading capabilities. The problem here is that processes interacting in such a context are bound to create deadlock situations; these processes may not afford to wait indefinitely on a pmessage invocation because that would prevent incoming messages from being serviced promptly (causing them to probably time-out). Currently, this is handled in IOC through short time-out values, and by tweaking the communication parameter representing the "number of retries for deadlock avoidance". When the pmessage times out, all pending incoming requests are processed by Liaison and the original pmessage is retried immediately, in the hope of removing the deadlock situation before giving up and raising a time-out error<sup>1</sup>. Pmessages accessed under this context must be made idempotent, because it is not possible to ensure the at-most-once semantics of connection-oriented protocols.

As a general rule, the choice of the transport protocol reflects whether the pmessage's implementation is idempotent or not. UDP is preferred for idempotent pmessages since it is more efficient than the more reliable TCP protocol.

## 4.3 Error Handling

Because errors may arise during network communication, it is necessary for the client invoking a pmessage to know whether or not the message has successfully been processed. The PmessageStatus class has been defined in the IOC library to help developers obtain all details on how a pmessage invocation performed.

Three types of errors are reported through the PmessageStatus class:

1. Communication error

These are the errors currently reported by the RPC library when communication fails.

2. Liaison error

---

<sup>1</sup>The loop is repeated up to the number of times specified in the "retriesLimit" parameter.



This error is reported when the primary copy of the Cobject does not exist in the server process (Liaison does not have a mapping for the Cobject).

### 3. Pmessage error

This is simply an error condition reported by a pmessage itself. By using this error reporting mechanism, the pmessage does not need to define specific return code and/or message parameters to report error conditions from the execution of the member function.

The following example shows a typical example of PmessageStatus on both client and server sides when the pmessage returns an error.

#### Client-side

```
PmessageStatus pStatus;
...
//
// Handle pStatus
//
pStatus = tcobj.suspend(10);
if (!pStatus.success()) {
    if (pStatus.pmessageError()) {
        // code to handle an error specific to the pmessage
        ...
    } else if (pStatus.liaisonError()) {
        // code to handle a Liaison error
        ...
    } else if (pStatus.rpcError()) {
        // code to handle an RPC error
        ...
    }
}
...
}
```

#### Server-side

```
...
//
// set Pmessage Error
// Error code is INVALID_CODE, error message stored in 'msg'
//
pStatus.setPmessageError(INVALID_CODE, msg); //
return pStatus;
...
}
```

## 5. Using the IOC library

This section briefly describes the steps a programmer goes through to implement a class of “Communicating” objects and then illustrates this with the use of an extended example.

A class that wants to support remote communication must be derived from class Cobject. The constructors for the class must support the creation for both primary and secondary copies. Member functions that are not required to be delivered to the primary copy (i.e. ones that the secondary copy itself can execute locally) are declared as usual in the public section of the interface.

However, the member functions that do need to be delivered to the primary copy for execution (pmessages) are **not** declared as usual in the public interface. Instead, they are declared in the class remote interface file (.ioc suffix) which is processed by the iocgen protocol compiler. Iocgen generates the proper client-side and server-side stubs that transparently handle communication with the remote object. The actual member function that implements a pmessage must have the word Primary appended at the end of the function's name. This is required to differentiate the pmessage's implementation from the generated client side stub which uses the function signature itself, so the client need not be concerned about whether a particular member function results in communication.

Consider a class "Machine" that encapsulates information and services offered by a typical workstation. The interface Machine.h would look something like:

```
class Machine {
public:
    // Constructors & Destructors
    ...

    // Static information
    String getHostname;      // hostname
    int getRAM();            // amount of RAM installed
    String getCPU();         // CPU type, eg "sun4/260"
    int getSwap();           // amount of Swap space on machine

    // Dynamic information
    float getLoadAverage(); // load average of the machine
    int getUserProcesses(); // number of user processes
    int getUsers();         // number of users
    int getFreeMemory();    // amount of free memory

    // arbitrary services
    void beep();             // emit a beep on the console
    ...
private:
    ...
}
```

Assume that in our implementation, objects of this class obtain the static information upon creation and store it as private state; thus requests for static information can be satisfied by returning pieces of this state. However, to satisfy requests for dynamic information, an object must each time obtain the information from the kernel and return it to the caller.

This class would be used as follows:

```
m = new Machine;
cout << "My load average is " << m->getLoadAverage() <<
      " and I have " << m->getUsers() << " logged on" << endl;
cout << "Here's my impression of the Road Runner..." << endl;
m->beep();
m->beep();
```

Obviously objects of class Machine can be accessed only from the address space in which they exist. But by using IOC, we can make this object available to other address spaces and machines.

Since IOC uses the primary/secondary copy paradigm described earlier, we must first decide which messages will be serviced by the secondary copy itself, and which messages must be delivered to the primary copy. Brief thought reveals that getHostname, getRAM(), getCPU(), and



getSwap() could be serviced by the secondary copy since that information is static; we decide that the rest of the messages must be delivered to the primary copy, i.e., they will be “pmessages”. We remove the signatures of the pmessages from Machine.h and put them in file Machine.ioc. Iocgen will parse Machine.ioc and generate the client and server side stubs.

So we now have two files “Machine.ioc” and “Machine.h” as follows:

#### Machine.ioc

```
FUNCTION void syncStatic()
RPCRESULTMEMBERS String hostname, int RAM, String CPU, int swap
FUNCTION float getLoadAverage()
FUNCTION int getUserProcesses()
FUNCTION int getUsers()
FUNCTION int getFreeMemory()
FUNCTION void beep()
```

#### Machine.h

```
class Machine : public Cobject {
public:
    // Constructors & Destructors
    Machine(CobjectType type=primary);
    Machine(ObjectId oid);
    ...

    // Static information
    String getHostname;    // hostname
    int getRAM();          // amount of RAM installed
    String getCPU();       // CPU type, eg "sun4/260"
    int getSwap();         // amount of Swap space on machine

    // pmessage member functions declarations generated by iocgen
    # include "MachineIocPublic.h"
    ...
private:
    ...
    // support member functions declarations for pmessages
    // generated by iocgen
    # include "MachineIocPrivate.h"
}
```

The member functions for the pmessages must be implemented in Machine.cc, with the word “Primary” appended to the function’s name. For example:

```
float getLoadAveragePrimary()
{
    // return value of load average
    ...
};
```

Note that initialization of the static information must now be performed through syncStatic() to ensure that both primary and secondary copies get the information. An object of class Machine can now be accessed from any remote ONC-connected address space as follows:

```

ObjectId oid1, oid2, oid3;

// Get ObjectIds of 3 remote Machine objects in oid1, oid2, oid3.
...

// Create secondary copies
Machine m1(oid1)1;
Machine m2(oid2);
Machine m3(oid3);

cout << m1->getHostname() <<
      " : RAM = " << m1.getRAM() <<
      "   load average = " << m1->getLoadAverage() << nl;
cout << m2->getHostname() <<
      " : RAM = " << m2.getRAM() <<
      "   load average = " << m2->getLoadAverage() << nl;
cout << m3->getHostname() <<
      " : RAM = " << m3.getRAM() <<
      "   load average = " << m3->getLoadAverage() << nl;

```

When we create `m1`, `m2`, `m3` with an `ObjectId` as the argument to the constructor, we automatically associate each of these local secondaries with a particular remote primary. Note that after creation, `m1`, `m2`, and `m3` are used as normal C++ objects. The message `getHostname()` is serviced by the local secondary copy, but the message `getLoadAverage()` is transparently delivered to the remote primary, executed there, the results marshalled back to the secondary, and then returned to the caller. To make a remote machine emit a beep, we can simply say:

```
m3->beep();
```

We could have further optimized this class by replacing all pmessages that return dynamic information with just one pmessage, `sync()`, which would copy the latest values of all dynamic information to the secondary. If this were done, then the functions `getLoadAverage()`, `getUserProcesses()`, `getUsers()`, and `getFreeMemory()` could then simply be regular member functions like `getRAM()` which return pieces of cached state.

```

FUNCTION sync()
RPCRESULTMEMBERS float loadAverage, int userProcesses,
                  int users, int packets

```

Since there is only a single primary copy of a `Machine` object in our server process (attribute `defaultReceiver` is `true`), and since each process typically has a well-known RPC program number, `ObjectIds` can often be easily built without having to obtain them from another service. Only the `hostNames` of the remote machines are needed. Also, the code can be further optimized by using a single variable in a loop to access all remote machines of interest, as seen below.

---

<sup>1</sup>We assume the constructor calls `syncStatic()`.

```

// Create secondary copy
Machine m(secondary)1;

// Query all machines listed in char** MachineList
for (machineName=MachineList; *machineName; machineName++) {
    ObjectId oid(*machineName, SERVER_RPCPROGNUM, machineClassId, 0);
    m.setObjectId(oid);
    m.staticSync();
    m.sync();

    cout << m->getHostname() <<
        " : RAM = " << m.getRAM() <<
        "   load average = " << m->getLoadAverage() << nl;
}

```

## 6. Discussion

IOC is to Object-Oriented programming what RPC is to procedural programming. The arguments for using IOC instead of RPC are similar to the generic arguments for using an object-oriented style instead of a procedural style, and are not discussed here.

IOC objects are passive. They exist within some remote process. This is different from some other object-oriented paradigms, like OTSO [Koi91], where objects can be thought of as remote processes themselves. In IOC, passive objects can emulate active objects by registering callback events with the object manager, Liaison. In addition, if a high degree of concurrency is necessary, these passive objects are free to spawn other processes upon instantiation, and act as the control point for these processes. Our belief is that this passive object paradigm with coarse support for active objects should be useful and sufficient for many distributed applications.

In our system, objects are stationary. In other words, once a primary copy of an object is created in an address space, it remains forever in that address space. Secondaries of that object can be obtained anywhere, but the primary doesn't move. This design decision was made because a *name service* is needed to keep track of moving objects. We chose not to design a name service of our own, or rely on nonstandard name services, because we wanted to make IOC available on a generic ONC platform. A name service commonly available on ONC is NIS (formerly "yellow pages"). However, this name service can only be used for relatively static information like usernames, encrypted passwords, hostnames, IP addresses, etc. Future name services from Sun, like NIS+, could be effectively used to keep track of moving objects.

Although a distinguishing feature of IOC is that local and remote objects behave identically from the client's point of view, IOC does support asynchronous (non-blocking) member function calls. Such calls pass the ObjectId of the client object to the possibly remote server object. To rendezvous back with the client, the server object uses this ObjectId to obtain a secondary copy of the client, and calls the appropriate member function on the client.

When IOC is used in single-threaded processes, all messages to objects in a particular process are serialized by the object manager of that process, and delivered one-by-one to the objects. Concurrency control is therefore implicit; real concurrency is not present. If IOC were to be used in a multi-threaded process, it would be desirable for class Cobject to provide primitives to lock or release objects, and queue messages to objects.

IOC's implementation provides hooks to support the automatic restart of processes that hold critical components of a distributed application (processes that provide an essential service). In our

---

<sup>1</sup>This constructor may not call syncStatic() since the ObjectId of the primary copy is unknown at creation time.

distributed application, we rely on NIS to identify candidate hosts where an essential process may be restarted, along with a synchronization mechanism that ensures that only one of these processes is active at any one time.

If the process containing a particular primary copy dies, then that primary copy is unavailable. IOC currently offers no support for persistence of objects, so if a primary copy becomes unavailable, the corresponding secondaries are crippled. We have experimented with primitive forms of persistence, but the most elegant solution would be to integrate IOC with a commercially available object-oriented database.

## 7. Conclusions

In this paper we have presented the IOC abstraction as a communication facility for distributed object-oriented applications, and its implementation as a C++ library. We have entirely used available technologies, principally SunOS RPC. IOC does not rely on low level facilities of the operating system. Therefore, it can easily be ported to other platforms. Even though our implementation is based on C++, other object oriented languages can be used equally effectively.

The IOC abstraction facilitates the design and implementation of distributed object-oriented applications by providing a simple paradigm for communication between remote objects which hides all communication details from the implementor. We have successfully used IOC in a major internal project.

## References

- [Alm85] G.T. Almes, A.P. Black, E.D. Lazowska, J.D. Noe, "The Eden system: A technical overview", IEEE Trans. Software Eng, Jan 1985.
- [Bir87] K.P. Birman, T.A. Joseph, "Reliable Communication in the Presence of Failures", ACM Trans. on comp. systs, Feb 1987.
- [Che87] David R. Cheriton, "The V Distributed System", CACM March 1988.
- [Cor91] John R. Corbin, "The Art of Distributed Applications", Springer-Verlag, 1991.
- [Cou88] George F. Coulouris, Jean Dollimore, "Distributed Systems — Concepts and Design", Addison-Wesley, 1988.
- [Koi91] Juha Koivisto, Juhani Malka, James Reilly, "OTSO — An Object-Oriented Approach to Distributed Computation", Usenix C++ Conference Proceedings, Washington, D.C., April 22-25, 1991.
- [Lis88] Barbara Liskov, "Distributed Programming in Argus", CACM March 1988.
- [Lyo84] B. Lyon, "Sun Remote Procedure Call Specification", and "Sun External Data Representation Specification", Sun Microsystems Inc. 1984.
- [Mul90] S.J Mullender, G.V. Rossum, A.S. Tanenbaum, A.V. Renesse, H.V. Stavern, "Amoe-  
ba — A Distributed Operating System for the 1990s", IEEE Computer, May 1990.
- [OMG91] OMG Consortium, "The Common Object Request Broker: Architecture and Specifi-  
cation", Document Number 91.8.1, August 26, 1991.

[TT91] Sun Microsystems, "Tooltalk (Beta) Programmer's Guide", Part number 800-6093-05 rev. 50, April 1991.



# Writing a Client-Server Application in C++

Paulo Guedes\*

Daniel Julin

OSF Research Institute  
1 Cambridge Center  
Cambridge, MA 02142  
pjpg@osf.org

School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA 15213  
dpj@cs.cmu.edu

## Abstract

*Applications based on the client-server model place a special emphasis on the specification of interfaces, the separation of interface and implementation and on the support for multiple implementations of the same interface. The class hierarchy of such an application has to be designed while taking these issues into account.*

*In this paper we present a model for writing client-server applications in C++, based in our experience with the Mach 3 multi-server system. Interfaces are defined by C++ abstract classes, from which implementations are derived. Implementations generally use multiple-inheritance to inherit functionality from other implementation classes.*

*We describe how this simple model was applied to the construction of the clients and servers that compose the Mach 3 multi-server, using standard C++. We discuss how multiple-inheritance simplified the design of the system and the need for run-time, type-safe pointer conversion. Finally, we give an overview of our class library and relevant aspects of the remote object invocation subsystem and summarize our experience with C++ in this project.*

## 1 Introduction

A major aspect of a system composed of cooperating clients and servers is how the interactions between them are specified and implemented.

Traditionally, the interface to a server is composed of a set of routines and is specified with an Interface Definition Language (IDL). A compiler, called the stub generator, generates the stubs that are linked to the clients, isolating their code from the details of the communication mechanisms.

In an object-oriented environment the natural evolution is to encapsulate the interface to the server in one or more classes, define their interface in a language-independent way with an IDL and generate stub classes to link with clients and servers. Several projects have implemented stub generators for C++[4, 11].

However, there are a number of limitations with this simple approach. One of the major concerns of a designer of a distributed application is the optimization of the interactions between clients and servers, as they amount to a very significant percentage of the costs

---

\* Author's current address: INESC/IST, R. Alves Redol 9, 1000 Lisboa, Portugal (pjpg@sabrina.inesc.pt)

involved. A common technique to deal with this problem is to use proxy objects[13]. Such objects are stubs with special hooks to allow the programmer to optimize or even avoid the communication between the client and the server while keeping the structure of the application clean.

Multiple implementations of the same interface are the rule and not the exception in client-server applications. There are at least always two implementations for each interface, the client-side and the server-side implementation, but often systems need a number of different implementations to handle various communication mechanisms.

Another concern is the extensibility of the system to allow a newer version of a server to be able to work with existing clients, and conversely to allow clients to work with old versions of the servers. Subtyping of object-oriented languages is a major tool to handle this problem, but it has to be used carefully and effectively. Moreover, the precise subtyping rules of the implementation language do not usually match those of the IDL.

In this paper we describe how we approach these issues in the Mach 3 multi-server system[5]. This system is composed of a number of servers running on top of the Mach 3 kernel, together with emulation libraries executing in the address space of the user programs. Together, they provide the complete services of an operating system. Both the servers and the emulation libraries are programmed as a set of C++ objects that communicate by invocation of C++ functions, independent of their location. Clients and servers are constructed from a library containing a number of common C++ classes. Both the interfaces and their implementation are specified in C++. Several interfaces have multiple client and server-side implementations. Servers may be modified without requiring changes to existing clients.

## 2 Interface and Implementation

This section describes how we specify the interfaces between clients and servers and how we support different client-side and server-side implementations of the same interface.

### 2.1 Interface

The interfaces between clients and servers are defined with C++ abstract classes. For the sake of clarity we shall ignore for the moment the details of the RPC subsystem and concentrate on the programmer's view of the C++ class hierarchy. When writing a server, the programmer must first design the classes that are externally visible by clients and then define the C++ abstract classes that constitute the interface. These abstract classes are the only server classes seen by the client's code. For example, if the server defines classes `naming` and `file` as:

```
class file {
    public:
        virtual int read(char*, int) =0;
        virtual int write(char*, int) =0;
};

class naming {
    public:
        virtual int open(char*, file**) =0;
};
```



a client would use them as follows:

```
file* f;
char buf[1024];
int error = fileserver->open("myfile", &f);
int bytesread = f->read(buf, 1024);
```

(assuming for the moment that variable `fileserver` is of type `naming` and has been previously initialized). The important thing to note is that this code is completely independent of the implementation of classes `naming` or `file`, of the mechanism used to communicate with the server and even of the location of the `naming` and `file` instances. The designer of the system has complete freedom to choose the location of the various objects and to use the communication mechanisms most appropriate to any given configuration. In particular, if the object being referenced happens to be co-located with the client, it is accessed directly as in a normal C++ program. The system can be conceived in terms of classes, or services, instead of being decomposed in address spaces and servers. That decomposition may be postponed to a latter phase, when servers are assembled out of the library of classes in the configuration that better fits the hardware or any other external constraints (e.g. security, performance).

## 2.2 Implementation

A client-side implementation class or proxy is always a subclass of the interface class that it implements. In the file example, a simple implementation that sends a RPC to the server for each request would be as follows:

```
class rpc {
    mach_port_t server_port;
public:
    virtual do_rpc(int method_id, ...) { /* some implementation */ }
};

class client_rpc_file: public virtual file, public virtual rpc {
public:
    virtual int read(char* buf, int len) {
        return do_rpc(method_id(read), buf, len);
    }
    virtual int write(char* buf, int len) {
        return do_rpc(method_id(write), buf, len);
    }
};
```

where class `rpc` is a generic class that implements remote procedure calls and uses `server_port` as the communication handle to talk to the server.

When the `open` function is called, the RPC subsystem creates an instance of an implementation class, `client_rpc_file` in this case, and returns it to the client. This is possible because `client_rpc_file` is a subclass of `client` and therefore can be used wherever a `client` is expected. Class `client_rpc_file` also uses inheritance to reuse the implementation of class `rpc`. Inheritance is used here with two of the three meanings described by [15], interface inheritance and implementation inheritance.

If there are multiple implementations of the same interface, the RPC subsystem has to decide which one to instantiate. In our system this is determined by the server, by sending the name of the client-side implementation that should be created.

Server-side implementations follow the same pattern: each implementation class is a subtype of the interface that it implements and also inherits code from the implementation class of the level above.

## 2.3 Why Multiple-Inheritance

The advantage of using multiple-inheritance becomes apparent when we consider *extending* the system by deriving new interfaces from the existing ones.

Consider, for example, that we define a new class `seek_file` that extends the interface of `file` with a new operation `seek`:

```
class seek_file: public virtual file {
public:
    virtual unsigned long seek(unsigned long, int) =0;
};
```

The client-side implementation of `seek_file` using class `rpc` would look like the following:

```
class client_rpc_seek_file: public virtual seek_file,
                           public virtual client_rpc_file {
public:
    virtual unsigned long seek(unsigned long position, int direction) {
        return do_rpc(method_id(seek), position, direction);
    }
};
```

The implementation class has to be a subtype of the interface class for the whole scheme to work, so deriving `client_rpc_seek_file` from `seek_file` offers no controversy. The debatable choice is whether it should also inherit from `client_rpc_file` or instead use it as a member. Let us look at this second alternative:

```
class client_rpc_file_SI: public file {
public:
    rpc parent_impl;
    virtual int read(char* buf, int len) {
        return parent_impl.do_rpc(method_id(read), buf, len);
    }
    virtual int write(char* buf, int len) {
        return parent_impl.do_rpc(method_id(write), buf, len);
    }
};

class client_rpc_seek_file_SI: public seek_file {
public:
    client_rpc_file_SI parent_impl;
    virtual unsigned long seek(unsigned long position, int direction) {
        return parent_impl.parent_impl.do_rpc(method_id(seek),
                                                position, direction);
    }
};
```

```

    virtual int read(char* buf, int len) {
        return parent_impl.read(buf, len);
    }
    virtual int write(char* buf, int len) {
        return parent_impl.write(buf, len);
    }
};

```

This second alternative basically forces us to repeat in each subclass all the methods of the base classes with trivial implementations as shown above. This may be acceptable for a small system with a reduced number of classes and methods, but is clearly undesirable in a large system with many classes and several levels of subtyping. Each time a new interface class is derived, its implementation class has to copy all the methods from the whole class hierarchy, each of them calling the implementation of the level above. Even worse, each time an interface class changes, because a parameter is added or deleted in a function, that change must be propagated by hand to all the implementation classes that derive from that interface. We feel that in practice this is equivalent to implementing multiple inheritance by hand, instead of letting the compiler do it for us. It is interesting to note that this example uses only single-inheritance in the interface classes and still leads to the use of multiple-inheritance.

## 2.4 Consistency Between Clients and Servers

The consistency between clients and servers is guaranteed by the lattice of interface classes. Changing an interface, by modifying an interface class, affects the implementations on both sides. During development this is easily detected with tools such as `make` that re-compile all the necessary files.

Traditionally systems have maintained consistency between clients and servers at the RPC level, requiring both sides to be modified if the messages exchanged between them change. We maintain this consistency at a higher level of abstraction that we call the service layer. Clients may remain unchanged as long as the interface classes are not modified, independent of the format of the messages exchanged with the server. If the message format changes, clients have to be re-linked (possibly dynamically) with new proxies, but their compiled code remains unchanged.

In some situations, we found it useful to define private interfaces between certain client and server side implementations. One example is the implementation of mapped files using the Mach external memory management facility. This implementation is fairly complex and requires close cooperation between the client proxy and the server object. By defining a private interface class visible only by these two implementations we are able to keep them consistent while still shielding the client's code from them.

A private interface class is just like an ordinary interface class, but it is visible only by the server and its proxies. It extends the public interface by defining new methods that are available only to the proxy. The proxy inherits the private interface and the implementation of the default proxy for the public interface.

## 2.5 Run-Time Pointer Conversion

In the previous example, a client that receives an object of type `seek_file` from `open` cannot use it as such unless the reference is converted to type `seek_file`. This case is of

special importance in our system because most operations start by looking up an object by name and then request operations on it. The type of the object is not known until it is looked-up in the directory, but depending on it different operations may be performed. For example, the system contains files, directories, symbolic links, mount points, pipes, various types of connection endpoints or sockets, all of which can be looked-up with the same operation, yet files and connection-less sockets require different protocols to read and write data.

For this reason, we use a type-safe run-time pointer conversion mechanism, similar to the one described in[2]. All classes contain a `castdown(class_id)` function that returns the pointer to the object cast to class `class_id` if `class_id` is a base class of the object (otherwise it returns 0).

Clients use it as follows: the name lookup routine returns a generic type that is the least common denominator of all possible types and an indication of the (interface) class of the object. Based on this information the client then calls `castdown` to convert the pointer to the real interface type of the object. For example, if a client had called `open` and an object of interface `seek_file` and implementation `client_rpc_seek_file` had been returned, the pointer would be converted from `file` to `seek_file` and used as such. A pointer conversion failure generally indicates an error in the operation of the system.

We only use such pointer conversions in the interface hierarchy. In the example above, the real type of the object is `client_rpc_seek_file`, but we convert it from the interface type we know at compile time (`file`) to the one we detect at run-time (`seek_file`).

We rejected the alternate method of defining all functions in the base class with an implementation that returns an error because it would force us to modify the base classes whenever a new class is added. In our case this is not even possible, as one of our requirements is to not change existing clients and servers when new services, and hence new classes, are introduced.

### 3 Overview of the Class Library

The class library contains the C++ classes from which servers are assembled. All operating system entities such as processes, files, sockets, are represented by objects and are implemented by these classes. The current prototype provides a self-hosting implementation of Unix BSD 4.3. Most of these classes were written from scratch, but a large amount of existing C code is reused (e.g. BSD Unix File System).

The hierarchy of interface classes is represented in Fig. 1. All classes have a common base `usItem` that defines a protocol available to all objects. The next level defines the interfaces for most of the objects. For example, `usName` is used for directories, `usByteIO` for files, `usRecIO` for connectionless sockets, `usTask` for process control, `usSys` for configuration management and `usEvent` for signals. Class `usNetbase` and all the other interface classes are used to describe the networking interfaces, which may be connectionless and record oriented (e.g. UDP sockets), connection oriented and record oriented, or connection oriented and byte-stream (e.g. TPC/IP sockets).

Interface class `usByteIO` describes the protocol for byte-stream input/output. It has four different server-side implementations for files, pipes, ttys and sockets. There are two different proxies for this interface, a simple proxy that sends messages for each operation and is used by pipes, tty and sockets, and a complex one that implements mapped files. From the client's point of view all these implementations are absolutely equivalent.

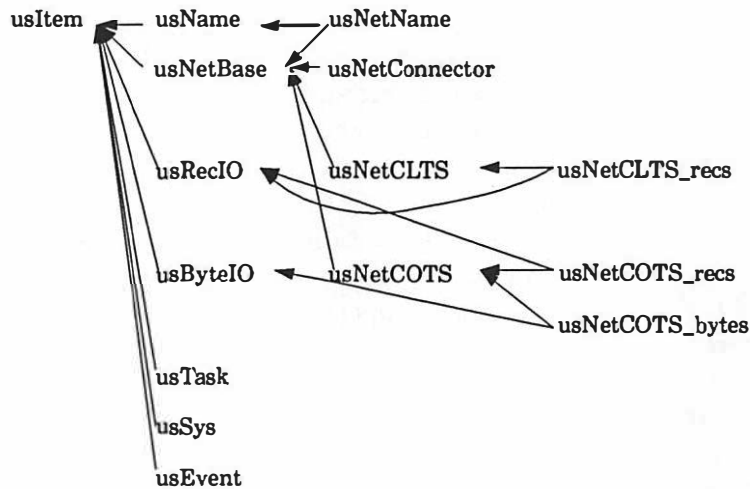


Figure 1: Interface classes.

Interface class `usName` defines a naming protocol to lookup objects by name, insert and delete them in a directory, mount one naming hierarchy on another and link two naming hierarchies. There are three server-side implementation classes that implement volatile directories, mount points and symbolic links, and several others that implement persistent directories in the Unix File System. Class `usName` is also a base class for the networking naming class `usNetName`.

Interface class `usTask` defines the protocol for process management. It currently has one server-side implementation and three proxies, the default one that sends messages for each operation and two others that cache certain characteristics of the process (e.g. pid, session identifier) to avoid contacting the server.

The complete description of all the interfaces and implementations is beyond the scope of this paper. We currently have the fifteen interface classes represented in the figure, eighteen proxy classes, about thirty server-side implementations and over twenty other classes internal to the servers. Several others are still being developed.

## 4 The Remote Object Invocation Mechanism

The remote object invocation subsystem consists of an RPC package extended with a layer that handles object references. An unusual characteristic of our RPC is that there are no compiled stubs. Instead, each function is described by a data structure that is parsed at run-time by a single pair of routines.

This section describes how the type information necessary to format messages is specified and the mechanics of transferring object references in messages, instantiating proxies and distributed garbage collection.

### 4.1 RPC Interface

The abstract classes described in the previous sections completely define the C++ interface between clients and servers. If no actual communication took place, they would provide

all the information necessary for clients and servers to interact. As this is not the case, we need to specify the RPCs exchanged between clients and servers.

In our library the abstract class defining an interface also specifies the RPCs that can be exchanged between a client of the class and the server implementing it. Each member function of an abstract class may correspond to a message. The abstract class must therefore contain the information describing the parameters to each member function so that messages can be generated. Implementation classes inherit these descriptions, so by default they are able to communicate through RPC. They may redefine or ignore this information if their message interface happens to be different.

Since we have no other tools available, we require the programmer to specify the RPCs using a set of macros and functions in both the interface declaration and implementation files. They generate a data structure with type information for the class in the form of a table containing for each function the function identifier, the description of the parameters and the address of the function. This type information is parsed at run-time by the RPC package to generate messages and to dispatch incoming calls. There are no compiled stubs in the traditional sense, instead a single pair of routines scan the type description of the function and pack or unpack the parameters accordingly.

The complete version of class naming presented in section 2 would now be:

```
// file naming.h
class naming {
    public:
        virtual int open(char*, file**) =0;
};
EXPORT_METHOD(open);

// file naming.cc
#include <naming.h>
DEFINE_ABSTRACT_CLASS(naming);
DEFINE_METHOD_ARGS(open, "rpc: IN string; OUT * object<file>;");
```

File naming.h contains the abstract class definition, as seen before, and the declaration of the functions exported to the RPC package. Macro EXPORT\_METHOD declares a structure with the method descriptor that uniquely identifies and describes function open in class naming. File naming.cc contains the description of the RPCs. Macro DEFINE\_ABSTRACT\_CLASS initializes the static member that contains the class' type information (e.g. class name, typeid used for pointer conversion). Macro DEFINE\_METHOD\_ARGS initializes the method descriptor for open with a string describing the type of its parameters. We still have to initialize the method descriptor with the address of the function and enter it in the per-class table. As functions may be redefined in subclasses, this cannot be done statically. Instead, implementation classes have a function init\_class that initializes all exported functions as follows:

```
// file naming_impl.h
#include <naming.h>
class naming_impl: public virtual naming {
    public:
        virtual int open(char*, file**);
};
```

```
// file naming_impl.cc
#include <naming_impl.h>
DEFINE_CLASS(naming_impl);
void naming_impl::init_class(void)
{
    // initialize base classes if any
    BEGIN_SETUP_METHOD_WITH_ARGS(naming_impl);
    SETUP_METHOD_WITH_ARGS(naming_impl, open);
    END_SETUP_METHOD_WITH_ARGS;
}
```

The call to macro `SETUP_METHOD_WITH_ARGS` initializes the various fields of the method descriptor for `open`: a per-process unique value to be used as identifier, a parsed version of the string that is easier and faster to manipulate at run-time and the pointer to the function. It also enters the method descriptor in the per-class function table, overriding any that might already be there. Base classes are initialized first, so that the function associated with the function identifier is the one of the most derived class, which is the desired behavior for virtual functions. For this reason we only support remote invocation of virtual functions.

A proxy for this class is defined as presented in section 2.2. The implementation of function `open` would be as follows:

```
int client_rpc_naming::open(char *name, file **f)
{
    return do_rpc(method_id(open), name, f);
}
```

Function `do_rpc` is defined in base class `rpc` and handles all outgoing messages. The first parameter `method_id(open)` is really the address of the method descriptor for `open`. `do_rpc` parses the method descriptor, extracts the parameters from the stack, constructs the message, performs the RPC, extracts the output parameters from the message and returns them on the stack. For the server-side the RPC does a similar set of operations. No server-side stubs are necessary since the description in the method descriptor is enough to unpack the incoming messages and pack the replies.

## 4.2 RPC Implementation

The remote invocation mechanism depends heavily on the underlying Mach 3.0 kernel[1]. The relevant Mach abstractions for this discussion are Mach ports, port rights and messages. Mach ports are unidirectional communication channels between processes. Port rights are capabilities allowing specific rights (e.g. send, receive) of access to a port. They have 32 bit names unique in an address space. Messages are typed collections of data passed between processes. Messages are sent to ports and can carry port rights, along with other basic types such as integers and characters.

Each object exported to clients is associated with a Mach port. The server that creates the object holds a receive right to that port, and each client that is given access to the object holds a send right to it. Initially, objects are created in the server with no associated ports. The remote invocation subsystem creates a Mach port for each object the first time that this object is returned as an output parameter in a remote invocation.

Clients send messages to the port that identifies the server object. A message contains a string specifying the name of the function being invoked, its parameters and a port indicating where the reply should be sent by the server. Simple types are passed by value. Objects are represented by a tuple composed of the Mach port associated with the object and the name of the proxy class to be instantiated at the destination if the object does not exist there.

Typically, messages sent from clients to servers pass basic types as input parameters and receive objects as output parameters. Let us use the `open` function as an example. The proxy class simply formats the `open` message using the information stored in the method descriptor and sends it to the port stored in its instance data (let us assume that it had been initialized to reference the directory service). On the server the object identified by the port is obtained from the table `port_to_object_table`, the function name is searched and its address is obtained. The stack frame is constructed as described earlier and function `open` at the server is called.

When this function returns, the remote invocation subsystem creates a port to represent the return parameter and uses a callback to store it in the object. It uses another callback to obtain the name of the proxy class and sends the tuple (`port`, `proxy class`) in the reply. It also stores the port in the table `port_to_object_table` for later lookups.

At the client, the message is scanned and the return frame is constructed. When the object reference is found, the (unique) port name is used to locate the object in table `port_to_object_table`. If the object exists, its pointer is returned; if not, the name of the proxy class is used to search another table called `class_map` and obtain a pointer to an object of the proxy class. This instance is used as a "factory" to create a new instance of the proxy class. The proxy object is initialized, storing the Mach port in its instance data, and its pointer is returned to the client. Table `class_map` is set-up statically at initialization time and contains one instance of each proxy class, associated with the class name.

When the client later invokes an operation on the proxy, it simply formats a message and sends it to the associated Mach port, repeating the process just described. The port representing the root of the directory service is obtained by clients at initialization time using a well-known port name.

The use of multiple-inheritance together with our run-time parser complicates this implementation because pointers to objects received as parameters have to be converted to the type specified by the interface. This conversion is done once again using the pointer conversion mechanism described previously. In the method descriptor, the description of the parameters contains the type to which the object should be converted. At initialization time we store there the `typeid` of that class and use it to perform the pointer conversion at run-time.

### 4.3 Garbage Collection

Garbage collection is greatly facilitated by the use of the Mach IPC system. Mach generates a notification message when there are no more send rights outstanding for the port, thereby allowing the server to destroy the associated object. Clients and servers garbage collect their objects locally using reference counting. When a Mach port is first associated with a server object its reference count is incremented to account for all remote proxies. When the reference count on a proxy object comes to zero, the associated Mach port is



deallocated and the proxy is destroyed. When the last port associated with the object is destroyed, Mach sends a notification message to the server that then decrements the reference count on the object. If it is the last reference the object is destroyed, otherwise the object will be deallocated when there are no more local references to it.

#### 4.4 Performance

The motivation to use a run-time parser to generate messages is to make each class as self-contained and lightweight as possible, so that clients and servers can be easily assembled from a class library without having to carry extra baggage. This is particularly attractive in a system composed of communicating objects where each class may be visible remotely.

The run-time parser is composed of a loop that for each parameter enters a switch statement depending on the type of the parameter. Each branch of the switch statement contains code similar to what a compiled stub would contain for the same type of parameter.

void f(int*) with MiG (simple server)	338 $\mu$ s
void f(int*) with MiG (complex server)	490-590 $\mu$ s
void f(int*) with C++ package	596 $\mu$ s
void f(file**) with C++ package	1156 $\mu$ s

Table 1: Remote invocation times measured on a 25MHz i386 HP Vectra with the client and the server on the same machine

Table 1 shows the total elapsed times for calling a simple C function returning one integer, for a similar C++ virtual function and for a C++ virtual function returning an object. The first row was obtained with a trivial client and server using stubs generated by the Mach Interface Generator (MiG). It represents the minimum time it takes to pack and unpack the parameters and perform the RPC. The C++ package is more complex since each invocation includes acquiring and releasing several mutexes, checking if there are still threads available to service other requests, checking the sequence number and other fields in the incoming message to prevent races relative to garbage-collection, and finally unpacking the message and performing the invocation (row 3). When similar operations are successively added to the MiG-based server, closer values are obtained (row 2). The last row shows the time to invoke a remote virtual function that returns an object. The additional costs in this case are the callbacks at the server, the longer time it takes Mach to send a message that contains a port, and the cost of instantiating the proxy object.

These values show that the performance of our RPC package is roughly equivalent to a MiG based system that does a similar amount of work, thus confirming our assumption that the cost of parsing the messages at run-time has a negligible effect on the overall performance. They also show that our C++ remote invocation subsystem is more complex and has more run-time overhead than a simple RPC based server, but the same level of complexity and overhead must be present in more realistic servers. It should be noted that these measurements are only approximate in the sense that it is very hard to compare systems that do different things.

Our 18 proxy classes have a total of 85 methods and consume about 70 Kilobytes of text space and initialized data. A MiG file with 87 similar definitions produces 30 Kilobytes for the user-side stubs and 63 Kilobytes for the server-side stubs. We were expecting more dramatic space savings by not using compiled stubs. Our proxies are larger than we expected because method `init_class` is generally very large and proxy classes contain several methods that have to be present in all classes of the library. In a 3.5K proxy, these two factors account for about 2K of the total text size. We never made any effort to optimize these values.

## 5 Related Work

The characteristics of our system forced us to put a much bigger emphasis on the support of different and customized client-side implementations of the same interface than most of the previous work on this field. This fact influenced heavily our main design choices. On the other hand, our system is more tightly integrated in the sense that our clients and servers are system entities that are not supposed to be coded or directly used by user programs. This allowed us to impose some restrictions that otherwise would have been undesirable, such as not considering mixing servers written in very different programming languages.

The Object Request Broker Architecture and Specification from OMG[10] provides a framework to define interfaces and specify the mechanisms by which objects can interoperate in a heterogeneous environment. The specification is language and even implementation independent and aims at being used by all kinds of applications. It defines an IDL that is a subset of C++ and supports a model for separating interface and implementation similar to[9]. We could have used their IDL and compiler to generate our abstract classes but they were not available when we started the project[3].

A number of other works addressed the issue of distributed C++ applications by proposing extensions to the language[14, 8]. In our case this was not an option. One of our requirements was the ability to compile the system with existing compilers, so that other people could reuse our work and we could not afford the effort of writing our own C++ pre-processor or compiler.

Extended C++[12] extends C++ with constructs for distributed programming. It is implemented as a pre-processor that generates standard C++ code. Our goal was not to define a distributed C++. Instead, we focused on using the language as it is to program a distributed system. The features in the language that are inadequate or irrelevant to our goals were simply not used. This allowed us to ignore all the hard issues about handling or disallowing certain constructs that do not apply to the distributed case.

Several systems have implemented RPC generators for C++ [4, 11] but they do not support multiple client-side implementations for the same interface.

The SOS project[13] introduced the notion of proxies and provided one of the earlier implementations of a system composed of a number of communicating C++ objects.

Choices[7] is another operating system written in C++. It provides a library of C++ classes that may be customized to construct different instances of an operating system. However, Choices is from the C++ point of view a monolithic application, i.e. it lives all in the same address space, not addressing the issues of client-server interaction.

## 6 Retrospective

This section evaluates the use of C++ in our project and some of the design decisions presented earlier.

### 6.1 Using C++

The Mach 3 multi-server was initially written in MachObjects[6], an object-oriented environment based on C macros and library routines that provides a programming model similar to Objective-C with some extensions. A large amount of BSD Unix C code is also reused with minimal changes.

The decision to use C++ instead of MachObjects was mostly motivated by the perceived need to use a well-known language instead of an arcane and virtually unknown environment like MachObjects.

Our major concerns with using C++ were the adequacy of static type checking in a dynamic system such as ours and the impact of having to use a different compiler.

The first problem was adequately addressed with our model for specifying interfaces and deriving the implementation classes and the additional mechanism for run-time pointer conversion. We think that one of the most positive aspects of using C++ was the better expressiveness of typed interfaces and the type-checking provided by the compiler.

The biggest challenge was the level of stability and maturity of the GNU C++ tools (compiler, debugger, linker) and their integration with our build environment. This is only a small part of the bigger problem of introducing a new compiler in an existing organization. We experienced many problems with poor or no support for virtual base classes and multiple-inheritance in the compiler and debugger. Some of these problems have been overcome with more recent versions of these tools; others have not.

### 6.2 C++ vs. an Interface Definition Language

One of the uncommon characteristics of this system is that it does not use an IDL and instead relies only on C++ to define RPCs.

There are two layers at which we could have used an IDL: the service layer where interfaces are specified in terms of classes and objects, and the RPC layer where we define the messages to be interchanged. In simple systems there is a one-to-one correspondence between these two layers, but in more complicated ones like ours that is not necessarily the case.

At the service layer our solution is basically to coalesce the IDL and C++ by defining the interfaces using the subset of C++ that “makes sense”. The main drawback of this approach is that we have no way to automatically check the constructs allowed by C++ that are invalid in the distributed case (e.g. pointers, public data). That task is left to the programmer, who has to carefully design these abstract classes using only a small subset of the language. In our system these interfaces are a crucial part of the design and their definition involved discussions between several people of different groups, which minimized the number of mistakes made by misuse of the language.

Defining the service interfaces with an object-oriented IDL and from there directly generating the C++ implementation classes does not support the notion of multiple client-side implementations for the same interface. Since there are no C++ classes that describe the interface, clients have to define variables of the C++ implementation types generated

by the IDL compiler, which defeats our goal of supporting different proxies for the same interface.

We could have used an object-oriented IDL to automatically generate the C++ abstract classes, instead of writing them manually as we did (if such IDL and compiler were available to us). A hierarchy of interface classes would be defined in IDL and the IDL compiler would generate our hierarchy of C++ abstract classes. One of the drawbacks of this approach is that it forces the designer of the system to define the interfaces in one language but then use the code generated by the IDL compiler to write the implementations. It might also be hard to establish a mapping between the subtyping rules of the IDL and those of C++. It has the advantage of having the IDL compiler checking constructs that make no sense in the distributed case and being more language independent, provided that the IDL compiler can generate more than one target language.

At the RPC level we require the programmer to specify the interfaces using macros basically because it was relatively easy to implement and allowed us to experiment with the run-time parsing of messages, but this is clearly unacceptable as a general solution.

A better solution would be to have an RPC compiler that automatically generates the necessary routines and data structures. Using C++ to specify the interfaces, as we did, keeps the system very homogeneous, but it is impossible to specify all the different parameter attributes supported by the communication subsystem, such as the in-line or out-of-line data and port manipulation options present in Mach IPC. A specialized IDL adequately addresses these problems, at the expense of introducing yet another language the programmer has to deal with.

The issue of interfacing clients and servers written in different languages can be solved at the message level by using the same RPC package on both sides and defining RPCs that are bit-for-bit compatible. At a certain point of our development a fileserver still written in MachObjects was perfectly able to communicate with C++ clients using the complex mapped files proxy, just because the RPCs between them had remained unchanged when the clients were converted to C++.

## 7 Conclusions

We present a model to write a client-server application in C++ that uses C++ abstract classes to specify the interfaces and multiple inheritance to construct the implementations. This model cleanly supports different client-side and server-side implementations of the same interface. The lattice of interface classes, composed of C++ abstract classes using mostly single inheritance, is visible to both clients and servers and guarantees the consistency of the implementations.

We believe that the model presented here to define the interfaces and to construct the implementations is a good way to write client-server applications in C++ and can be widely used in the C++ community.

## 8 Acknowledgements

Many people at CMU and at the Research Institute of OSF have participated in the design and implementation of the system. Beside the authors, they are: J. Mark Stevenson, Jonathan J. Chew, Paul Neves, Paul Roy, Robert Baron, Alessandro Forin, Jeffrey

Heller, Michael Jones, Keith Loepere, Douglas Orr, Richard Rashid, Franklin Reynolds and Richard Sanzi. John LoVerso, Franklin Reynolds and the anonymous reviewers provided helpful comments.

## References

- [1] Richard P. Draves. A Revised IPC Interface. In *Proceedings of Mach Usenix Workshop*, Burlington, Vermont, October 4-5 1990.
- [2] K. Gorlen, S. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.
- [3] P. Guedes. Use of Object-Oriented Technology in the Implementation of a Distributed Operating System. In *Addendum to the Proceedings of OOPSLA/ECOOP*, Ottawa, Canada, Oct 1990. Sigplan Notices Special Issue.
- [4] J. Heliotis and P. Mansey. Remote Object Invocation and its Implementation in C++. In *Proc. C++ at Work*, C++ Report and the Wang Institute of Boston University, 1989.
- [5] D. Julin, J. Chew, P. Guedes, P. Neves, P. Roy, and M. Stevenson. Generalized Emulation Services for Mach 3.0 - Overview, Experiences and Current Status. In *Proceedings of Usenix Mach Symposium*, Monterey, CA, Nov 20-22 1991.
- [6] D. Julin and R. Rashid. Machobjects. Technical report, Mach Project, Carnegie Mellon University, 1989.
- [7] P. Madany, D. Leyens, and V. Russo. A C++ Class Hierarchy for Building UNIX-like File Systems. In *Proc. 1988 Usenix C++ Conference*, Denver, CO (USA), Oct 1988.
- [8] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA 89*, New Orleans, 2-6th October 1989.
- [9] B. Martin. The Separation of Interface and Implementation in C++. In *Proc. 1991 Usenix C++ Conference*, Washington, DC (USA), Apr 1991.
- [10] OMG. The common object request broker: Architecture and specification. Technical Report OMG Document Number 91.12.1, Revision 1.1, Object Management Group, December 1991.
- [11] G. Parrington. Reliable Distributed Programming in C++: The Arjuna Approach. In *Proc. 1990 Usenix C++ Conference*, San Francisco, CA (USA), Apr 1990.
- [12] Robert Seliger. Extended C++. In *Proc. 1990 Usenix C++ Conference*, San Francisco, CA (USA), Apr 1990.
- [13] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th. International Conference on Distributed Computer Systems*, pages 198-204, Cambridge, Mass. (USA), May 1986. IEEE.
- [14] M. Tieman. Wrappers: Solving the RPC Problem in GNU C++. In *Proc. 1988 Usenix C++ Conference*, Denver, CO (USA), Oct 1988.
- [15] Jim Waldo. Controversy: The Case for Multiple Inheritance in C++. *Computing Systems*, 4(2), Spring 1991.



# Integrating the Sun Microsystems XDR/RPC protocols into the C++ stream model

Robert E. Minnear      Patrick A. Muckelbauer  
Vincent F. Russo  
Department of Computer Sciences  
Purdue University  
W. Lafayette, IN 47907  
email: {minnear|muckel|russo}@cs.purdue.edu

## Abstract

This paper reports our experiences integrating the Sun Microsystems RPC and XDR protocol specifications into the C++ model of input/output streams. As part of the *Renaissance* operating system project, we wish to construct network servers and clients, written in C++, which interoperate with existing UNIX clients and servers. We discovered that, although it would be possible to re-implement the procedural based XDR/RPC implementation distributed by Sun Microsystems in C++, it is far cleaner to integrate the protocols with the C++ I/O stream model. We feel the resulting model provides a cleaner way of implementing RPC clients and servers without losing functionality or compatibility with existing clients and servers.

## 1 Introduction

This paper reports our experiences integrating the Sun Microsystems (SUN) Remote Procedure Call (RPC) and External Data Representation (XDR) protocol specifications with the C++ model of I/O streams. This project was undertaken as part of the *Renaissance* operating system project at Purdue. *Renaissance* is an object-oriented operating system implemented in C++. Part of our work with *Renaissance* requires us to implement in C++ network servers and clients which interoperate with existing C/UNIX clients and servers supporting the SUN protocols. For example, we wish to communicate with both NFS[Sun88b] servers and clients. During our research, we discovered that although it would be possible to re-implement the procedural based XDR/RPC implementation distributed by Sun Microsystems in C++, it is far cleaner to integrate the protocol into the C++ I/O stream model. We feel the resulting model provides a cleaner and simpler vehicle for implementing XDR/RPC clients and servers without losing functionality or compatibility with existing clients and servers.

## 2 Background

For distributed applications to communicate, it is necessary to provide a mechanism to transfer data back and forth between them. A simple and popular model for this intercommunication is the *remote procedure call*[BN84]. The remote procedure call idea is based on a client/server model of computation. In this model, client programs on various nodes

in a network of machine make requests to server programs running on other nodes. The model presented to client programmers is quite simple: an ordinary procedure or function is invoked locally to obtain a service. Likewise, the implementor of a server program provides its services to the clients as a set of procedures or functions. The real work is connecting client calls to the server functions implementing the desired services. Since the server and client program may be running on different and possibly heterogeneous machines, normal procedure calls cannot be used. Instead, the remote procedure call model is based on the idea that the procedure the client program actually invokes is a "stub" function that packages the arguments into a network request and forwards them via the network to a peer stub running on the server machine. The peer stub unpackages the arguments, calls the server function locally, and packages the results into a network reply. Back at the client side, the results are unpackaged by the stub and returned to the client program as part of the ordinary procedure calling sequence. The remote procedure call model, therefore, provides the illusion of accessing services by local procedure calls, while hiding the details of network processing and argument/result marshalling/unmarshalling from client and server programmers.

There are two major issues which arise when designing and implementing a remote procedure call system: how are the stubs at both ends constructed, and what mechanism do the stubs use to communicate? This paper is not concerned with the first issue. Currently, we resort to writing the stubs by hand. Instead, the rest of the paper focuses on stub-to-stub communications. The issue of peer communication consists itself of numerous sub-issues including: what low level transport mechanism should be used for communication, how are remote services named and located, and how are differences in data representation between clients and servers resolved? As a solution, we chose to use the specifications from Sun Microsystems for remote procedure call and external data representation[Sun85a, Sun88a].

## 2.1 SUN-RPC

The SUN Remote Procedure Call Specification (SUN-RPC) defines a remote procedure call based communication mechanism where server programs register their services and wait at network ports for requests from client programs. A per-machine server called the "port-mapper" listens at a well-known network port and responds to queries about the location of other servers on that machine. New servers simply have to register themselves with the port-mapper. Once a client has located the network port of a server, it can call procedures within that server. Each call from a client to a server specifies a procedure number to call along with the arguments to that procedure. The primary purpose of the SUN-RPC specification is to standardize the representation of all this information inside network packets destined to/from the network port on which the server is listening.

SUN-RPC is flexible in regards to transport protocols, it can use either a connectionless unreliable datagram interface or a fully connected reliable byte-stream. SUN-RPC does not specify any semantics in regards to reliability, this is left up to the choice of transport protocol and the particular implementation of SUN-RPC. To provide reliability over an unreliable transport layer, an implementation could use timeouts and retransmissions. No additional effort is necessary for reliable transport layers. Although, in general using a reliable transport layer incurs a significant expense (i.e setting up connections, etc.).



## 2.2 External Data Representation

Every machine has an internal representation for data. When two heterogeneous machines wish to communicate, they must be able to interpret each other's data properly. One approach requires all data to be tagged with an architecture descriptor that can be decoded by another machine and used to determine how to decode the data into its internal representation. The SUN-RPC protocol instead chooses to define a canonical representation for data, requiring every host to translate its internal representation of data to and from the canonical representation[Sun85a]. The External Data Representation (XDR) protocol[Sun88a] is the specification of this canonical representation. XDR is a description language for data representation, and is independent of transport layer. XDR defines a representation for a set of basic types (int, float, string, etc.) and rules for representing more complex structures as composites of the basic types. For example, it is possible in XDR to canonicalize a singly linked list.

The representation provided by the XDR specification does not explicitly include any type information. Rather, it assumes both ends have agreed on the type of the data to be exchanged. This has the advantage of being both less complicated and more space efficient. The drawback is there must exist some other mechanism to agree upon the types of the data to be exchanged. In SUN-RPC, for instance, a message is comprised of a header and data. The header provides the information to uniquely identify a remote procedure and thus defines how the data are to be interpreted. The header is itself encoded into XDR format, but both sides have implicitly agreed on its representation.

Existing SUN-RPC implementations require the user to provide to each RPC call XDR encode and decode routines, arguments, and memory for results. The XDR routines are subsequently called by the SUN-RPC layer and not by the user, implying that the XDR routines must retrieve all of the resultant data at one time. Further, the decoding of dynamic data structures can lead to implicit memory allocations.

The routines which encode and decode data in their XDR format can either be written by an automatic generator which takes as input type specifications[Sun85b], or by hand.

## 2.3 Problems with existing implementations

One drawback of the current implementation of SUN-RPC that we see is the lack of type safety. The lack comes from the standard way in which SUN-RPC calls are made. Each call specifies functions to perform the XDR encoding and decoding of the data along with a pointer to the data storage. In the same way the C library `printf` function cannot check type compatibility between the pointer to its arguments and its format string, the RPC call functions cannot check that a pointer to an object of the proper type has been associated with the XDR function.

Another disadvantage we see is that the standard way of encoding and decoding data requires all the arguments to be present at encode time and all the results to be decoded at once. For example, when decoding a linked list, it is necessary to allocate space for all of the list nodes and copy each out of the received network buffer, rather than being able to consume one list node from the buffer at a time.

The remainder of this paper will discuss the C++ stream I/O model and present how we have recast SUN-RPC and XDR in terms of this model in order to address these problems.

### 3 C++ Streams

In C++, I/O is provided through the *stream* abstraction[Str86, Sho89]. Streams in C++ provide a user extensible, type-safe, efficient, and flexible mechanism for input and output. We view C++ streams as translators, which take typed objects and convert them to and from sequences of bytes.

In the standard C library, I/O is provided by the `scanf/printf` functions. These routines parse an argument format string searching for type specifiers which explicitly type the remaining arguments. However, there is no static, compile time type checking done for the argument list. Type mismatches between the format list and the actual parameters cause runtime errors or erroneous output. C++ streams avoid both the overhead of parsing the format string, and the type-insecurity by overloading the `<<` and `>>` operators for builtin and user-defined types. This allows the compiler to select the proper function based on the type of the argument in the stream at compile time. If a programmer changes the type of an argument, the correct overloaded operator implementation is chosen when the program is recompiled. Good software engineering dictates that errors should ideally be caught as early in the software development cycle as possible.

In much the same way as XDR, C++ streams provide operators for translating a set of basic types. Operators on composite structures may be formed by combining calls to the basic operators. User defined structures may be integrated into the stream model by overloading the `<<` and `>>` operators. This allows these structures to appear syntactically similar to the basic types. There is no equivalent mechanism in the standard C I/O libraries. The only way to achieve this in C would be to allow the user to define type specifiers that may appear in the format string. No currently known implementations support this ability.

### 4 C++ stream model solution

The conceptual simplicity of the C++ stream model led us to rethink XDR in terms of streams. It seemed only natural to recast XDR data translation in terms of the C++ `<<` and `>>` stream operators.

Specifically, our goal is to develop a stream model derived from the C++ stream model that incorporates the functionality of both the XDR protocol and other typical and familiar streams, such as ASCII streams. The result is a new model which provides similar functionality and includes all the benefits of the existing C++ stream model. In addition, it provides the flexibility necessary to model the XDR protocol.

Our first step was to redefine the high level abstraction of streams so that it was possible to incorporate the XDR protocol into it. Currently, C++ streams are modeled as a set of output streams and input streams capable of converting typed data to and from ASCII format respectively. Our top level abstraction views streams as translators, responsible for processing data before passing it to its ultimate destination. C++ streams are subsumed in our model as ASCII translators. The XDR protocol can, likewise, be incorporated into our model.

We next observed that, while typical ASCII streams tend to read and write files, XDR streams, primarily used in creating network packets, tend to read and write buffers. Our stream model had to accommodate such disparate usages. Fortunately, the C++ stream model provides a good example of how to deal with such a problem and our solution is very similar. By viewing the functionality of a stream as a dichotomy comprised of a "how"

part, responsible for data translation, and a “where” part, responsible for data transport, the needed flexibility can be gained.

In our model, output streams are implemented by subclasses of the abstract *OutputStream* class. *OutputStreams* are responsible for defining the “how” part, but not responsible for defining the “where” part, of a stream. An object of type *ByteSink* is passed to the constructor of an *OutputStream* for the purposes of doing the “where” part. A *ByteSink* is a simple abstract class defining operations to write data to the sink (*write*) and to flush any buffer the sink may have (*flushBuffer*).

Likewise, input streams are implemented by the abstract *InputStream* class. A *ByteSource* is supplied to the constructor of *InputStream* to supply the “where” part. A *ByteSource* is an abstract class defining a single method (*read*), responsible for providing raw data. This decoupling of “how” and “where” allows for the greatest degree of flexibility. Any *ByteSink* can be associated with an *OutputStream* to produce an output stream, and likewise for input streams. For further clarity, the declarations on these four top level abstract classes are shown below.

```
class ByteSink {
public:
    virtual int write( const char *, int ) = 0;
    virtual void flushBuffer() { return; }
};

class OutputStream {
protected:
    ByteSink * _bs;
    ...
public:
    OutputStream( ByteSink & );

    /* conversion operators for basic types */
    virtual OutputStream & operator << ( char ) = 0;
    virtual OutputStream & operator << ( unsigned char ) = 0;
    virtual OutputStream & operator << ( short ) = 0;
    virtual OutputStream & operator << ( unsigned short ) = 0;
    virtual OutputStream & operator << ( int ) = 0;
    virtual OutputStream & operator << ( unsigned int ) = 0;
    virtual OutputStream & operator << ( long ) = 0;
    virtual OutputStream & operator << ( unsigned long ) = 0;
    virtual OutputStream & operator << ( const char * ) = 0;
    virtual OutputStream & operator << ( const void * ) = 0;
    virtual OutputStream & operator << ( asVarOpaque & ) = 0;
    virtual OutputStream & operator << ( asFixedOpaque & ) = 0;

    /* flush any underlying ByteSink buffer */
    OutputStream & flushBuffer() { _bs->flushBuffer(); return( *this ); }
    ...
};

class ByteSource {
public:
    virtual int read( char *, int ) = 0;
};

class InputStream {
protected:
    ByteSource * _bs;
```

```

    ...
public:
    InputStream( ByteSource & );

    // conversion operators for basic types
    virtual InputStream & operator >> ( char & ) = 0;
    virtual InputStream & operator >> ( unsigned char & ) = 0;
    virtual InputStream & operator >> ( short & ) = 0;
    virtual InputStream & operator >> ( unsigned short & ) = 0;
    virtual InputStream & operator >> ( int & ) = 0;
    virtual InputStream & operator >> ( unsigned int & ) = 0;
    virtual InputStream & operator >> ( long & ) = 0;
    virtual InputStream & operator >> ( unsigned long & ) = 0;
    virtual InputStream & operator >> ( char * ) = 0;
    virtual InputStream & operator >> ( asVarOpaque & ) = 0;
    virtual InputStream & operator >> ( asFixedOpaque & ) = 0;
    ...
};

```

Concrete I/O streams are created by subclassing `OutputStream` and `InputStream` and implementing the abstract operators for converting the basic types. For example, we provide an `ASCIIOutputStream` and `ASCIIInputStream` which provide the normal terminal I/O processing.

A small example should demonstrate some of the flexibility of our model. Assume you are provided with *ASCIIOutputStream*, a subclass of `OutputStream`, that translates typed data to ASCII strings, and a *BufferByteSink*, a subclass of `ByteSink`, whose constructor is supplied a buffer and a length to be used for storing subsequent writes. Given these two classes, it is trivial to implement the routine `IntegerToString()` which converts an integer to its ASCII string representation.

```

void
IntegerToString( int n, char * buf, int length)
{
    BufferByteSink bs( buf, length );
    ASCIIOutputStream aos( bs );
    aos << n;
}

```

## 4.1 XDR

To integrate the XDR protocol into our model, we provide the *XDROutputStream* and *XDRInputStream*, subclasses of `OutputStream` and `InputStream`. `XDROutputStream` implements the `OutputStream`'s abstract `<<` methods on the basic types by translating the argument to XDR format before writing data to its `ByteSink`. Likewise, the `XDRInputStream` implements the `InputStream`'s abstract `>>` methods on basic types by translating the data read from the its `ByteSource` from XDR format to host format. `XDROutputStream` and `XDRInputStream` provide the basis for all XDR format conversion and most of the functionality provided by SunOS's procedural based XDR implementation.

The benefits of our implementation of XDR versus that of traditional implementations is mostly attributable to the benefits inherited from the stream model: type-safeness, flexibility, and extensibility. By overloading the `<<` and `>>` operators, XDR input and output streams can be extended to understand user defined objects. The ability of supplying

user defined ByteSinks and ByteSources to the XDROutputStream and XDRInputStream streams, affords the user greater accessibility to the XDR protocol. This is in contrast to SunOS XDR implementation, where the XDR model comprises both translation and destination of data, limiting its use to only that which is provided. We believe that the stream model is cleaner and easier to understand and use than Sun's procedural based model. The example in section 4.3 should help verify these points.

It should be noted that several lessons were learned by collapsing two seemingly unrelated ideas/models into a single unified model. First, it makes one develop sufficiently powerful and flexible abstractions to encompass both ideas/models into a single model. Second, the benefits of one model can now be shared with another. This was particularly important with the XDR protocol, which gained all the benefits of the stream model.

## 4.2 RPC

Much of the design of our RPC model is based on our experiences with the SunOS implementation of XDR/RPC. For the most part we model Sun's interface, except in one important area: XDR encoding and decoding of arguments and results. As mentioned earlier, the SunOS implementation of RPC requires callers to provide encoding and decoding routines to the low level RPC call routines. The invocation of these XDR routines is done by the RPC layer itself. This implies that all encoding and decoding must be performed at once. Because the size of the return value can vary (variable length arrays for instance) many users of RPC rely on the implicit memory allocation of the XDR layer. We found this very error prone as programmers of long lived servers often tended to forget to free this memory. In our model, the RPC layer expects the arguments to be in XDR format and returns the results in XDR format. A benefit of our solution is the ability to return a pointer into the network packet, removing an unnecessary copy, and allowing arguments to be shifted off (decoded) incrementally by the user. This scheme reduces or eliminates most memory allocation requirements. It should be noted that Sun's implementation can simulate our model of sending and receiving data in XDR format by simply using encode and decode routines which copy data but do not translate it. However, this does incur the cost of copying the data to and from the network packets. For further clarity, the partial declarations used to implement our RPC model are shown below.

```
class RPCClient {
protected:
    ...
public:
    RPCClient( IPAddress & host, unsigned int program,
               unsigned int version, unsigned int remotePort = 0,
               unsigned int localPort = 0 );
    RPCClient( char * host, unsigned int program,
               unsigned int version, unsigned int remotePort = 0,
               unsigned int localPort = 0 );
    ...
    ByteSource * call( unsigned int proc, ByteSource * args = 0 );
    ByteSource * call( unsigned int proc, char * args, int length );
    ByteSource * nextResponse( int timeout = RPCTimeout );
};

class RPCServer {
protected:
    ...
};
```

```

public:
    RPCServer( unsigned int program, unsigned int version,
               unsigned int protocol, unsigned int localPort = 0 );
    ***
    void setAcceptSuccess();
    void setAcceptProcUnavail();
    void setAcceptGarbageArgs();
    void setReplyAuthNull();
    int  getCallMessage();
    int  checkCall();
    void sendReply();
};

```

### 4.3 Example

The following example contrasts the style between the current SunOS XDR/RPC implementation and our implementation. This example implements, in each model, an RPC server that provides a client/server function similar to the UNIX `finger` program.

First we present the client and server programmed using the standard SunOS library linked with C++.

```

// common_model.h

#define RFINGERPROG      400000
#define RFINGERVERS      1

#define NULLPROC         0
#define ONEUSERPROC      1
#define ALLUSERSPROC     2

```

These constants are common to both models.

```

// rfinger.h:

struct loginfo {
    char tty[ 8 ];
    char host[ 16 ];
    long loginat;
    long idletime;
    char writable;
    loginfo * next;
};

struct person {
    char logname[ 8 ];
    char * realname;
    char * office;
    char * officephone;
    char * homephone;
    long maillastrecv;
    long maillastread;
    char loggedin;
    loginfo * logins;
};

```

```

    person * next;
};

```

The server returns user information as a singly linked list of *person* structures. Each person structure itself contain a singly linked list of *loginfo* structures which containing information about the each of the user's logins.

```
// common.cc
```

```

bool_t
xdr_loginfo( XDR * xdrs, loginfo * login )
{
    char * tmp = login->tty;
    int size = 8;

    if ( ! xdr_bytes( xdrs, &tmp, &size, 8 ) ) return( FALSE );
    size = 16;
    tmp = login->host;
    if ( ! xdr_bytes( xdrs, &tmp, &size, 16 ) ) return( FALSE );
    if ( ! xdr_long( xdrs, &login->loginat ) ) return( FALSE );
    if ( ! xdr_long( xdrs, &login->idletime ) ) return( FALSE );
    if ( ! xdr_char( xdrs, &login->writable ) ) return( FALSE );
    if ( xdrs->x_op == XDR_DECODE ) login->next = 0;
    return( xdr_pointer( xdrs, &login->next, sizeof( loginfo ), xdr_loginfo ) );
}

```

```

bool_t
xdr_person( XDR * xdrs, person * who )
{
    char * tmp = who->logname;
    int size = 8;

    if ( ! xdr_bytes( xdrs, &tmp, &size, 8 ) ) return( FALSE );
    if ( xdrs->x_op == XDR_DECODE ) who->realname = 0;
    if ( ! xdr_wrapstring( xdrs, &who->realname ) ) return( FALSE );
    if ( xdrs->x_op == XDR_DECODE ) who->office = 0;
    if ( ! xdr_wrapstring( xdrs, &who->office ) ) return( FALSE );
    if ( xdrs->x_op == XDR_DECODE ) who->officephone = 0;
    if ( ! xdr_wrapstring( xdrs, &who->officephone ) ) return( FALSE );
    if ( xdrs->x_op == XDR_DECODE ) who->homephone = 0;
    if ( ! xdr_wrapstring( xdrs, &who->homephone ) ) return( FALSE );
    if ( ! xdr_long( xdrs, &who->maillastrecv ) ) return( FALSE );
    if ( ! xdr_long( xdrs, &who->maillastread ) ) return( FALSE );
    if ( ! xdr_char( xdrs, &who->loggedin ) ) return( FALSE );
    if ( xdrs->x_op == XDR_DECODE ) who->logins = 0;
    if ( ! xdr_pointer( xdrs, &who->logins, sizeof( loginfo ),
        xdr_loginfo ) ) return( FALSE );
    if ( xdrs->x_op == XDR_DECODE ) who->next = 0;
    return( xdr_pointer( xdrs, &who->next, sizeof( person ), xdr_person ) );
}

```

```

bool_t
xdr_personptr( XDR * xdrs, person * who )
{
    return( xdr_pointer( xdrs, &who, sizeof( person ), xdr_person ) );
}

```

```

void
printloginfo( loginfo * info, int active ) { ... }

void
printperson( person * who )
{
    ...
    loginfo * info;
    info = who->logins;
    while ( info != 0 ) {
        printloginfo( info, who->loggedin );
        info = info->next;
    }
    ...
}

```

These routines provide the XDR encoding/decoding operations and print functions for the structures. These routines illustrate some of the problems we see with the Sun implementation. The xdr encoding/decoding functions seem unnecessarily complex. The routines also depend on the implicit memory allocation supplied by `xdr_wrapstring` and `xdr_pointer`. Also, the additional routine `xdr_personptr`, to be supplied to subsequent RPC calls, is required to encode/decode the entire structure. The memory allocation problems stems directly from the consequence of the Sun implementation of RPC requiring all data to be processed at one time.

```

// client.cc:

int
main( int argc, char * argv[] )
{
    int i;
    char * host = /* parse argv */;
    char * user = /* parse argv */;

    if ( user == 0 ) {
        struct person first;
        struct person * item;

        i = callrpc( host, RFINGERPROG, RFINGERVERS, ALLUSERSPROC,
                     xdr_void, 0, xdr_personptr, &first );

        if ( i == 0 ) {
            item = &first;
            while ( item != 0 ) {
                printperson( item );
                cout << "\n\n";
                item = item->next;
            }
        }
        else {
            cerr << "bad user name\n";
        }
    }
    else {

```



```

        struct person who;

        i = callrpc( host, RFINGERPROC, RFINGERVERS, ONEUSERPROC,
                     xdr_wrapstring, &user, xdr_personptr, &who );
        if ( i == 0 ) printperson( &who );
        else cerr << "bad user name\n";
    }
}

```

Note that the callrpc routine is required to take encode and decode routines for processing the arguments and results. In this implementation, the argument data must be copied twice, once during the construction of the data to be passed to the callrpc routine and again during the encoding processes into the network packet.

```

// server.cc:

int
getpersonbyname( char * name, person * who ) { ... }

person *
getallpersons() { ... }

void
dispatch( svc_req * request, SVCXPRT * xprt )
{
    char * user = 0;
    person who;
    person * list = 0;
    int i;

    switch ( request->rq_proc ) {
        case NULLPROC:
            svc_sendreply( xprt, xdr_void, 0 );
            return;

        case ONEUSERPROC:
            svc_getargs( xprt, xdr_wrapstring, &user );
            i = getpersonbyname( user, &who );
            if ( i == 0 ) svcerr_decode( xprt );
            else svc_sendreply( xprt, xdr_personptr, &who );
            return;

        case ALLUSERSPROC:
            list = getpersons();
            svc_sendreply( xprt, xdr_personptr, list );
            return;
    }
    svcerr_weakauth( xprt );
}

main( int argc, char * argv[] )
{
    int err;
    SVCXPRT * xprt;

    // Create a transport handle to get requests and send replies on.

```

```

    if ( ( xprt = svcudp_create( RPC_ANYSOCK ) ) == 0 ) exit( ... );

    // Unmap any old registration and register server with portmapper.
    pmmap_unset( RFINGERPROG, RFINGERVERS );

    if ( ! svc_register( xprt, RFINGERPROG, RFINGERVERS, dispatch,
        IPPROTO_UDP ) ) exit( ... );

    svc_run();
    exit( 1 );
}

```

The routines `getpersonbyname` and `getallpersons` are responsible for obtaining all the necessary data for the result, and in particular, constructing the linked list to be returned. It is typical of such routines to rely on the implicit memory allocation techniques of the SunOS XDR implementation.

Because the server is responsible for buffering all resultant data prior to sending, two copies are necessary. One to construct the resultant data and one to encode it into the network packet.

Next we present the above example re-implemented with our new stream model.

```

// rfinger.h

class Loginfo {
public:
    char _line[ 8 ];
    char _host[ 16 ];
    long _loginat;
    long _idletime;
    int _writable;
    InputStream & shiftRight( InputStream & is );
    OutputStream & shiftLeft( OutputStream & is );
    void inspect( OutputStream & os );
};

class Person {
public:
    char _name[ 8 ];
    char _realName[ 32 ];
    char _office[ 32 ];
    char _officePhone[ 32 ];
    char _homePhone[ 32 ];
    long _mailLastReceived;
    long _mailLastRead;
    int _loggedin;
    InputStream & shiftRight( InputStream & is );
    OutputStream & shiftLeft( OutputStream & is );
    void inspect( OutputStream & os );
};

inline InputStream &
operator >> ( InputStream & is, Loginfo & loginfo )
{
    return( loginfo.shiftRight( is ) );
}

```

```

inline OutputStream &
operator << ( OutputStream & os, Loginfo & loginfo )
{
    return( loginfo.shiftLeft( os ) );
}

inline InputStream &
operator >> ( InputStream & is, Person & person )
{
    return( person.shiftRight( is ) );
}

inline OutputStream &
operator << ( OutputStream & os, Person & person )
{
    return( person.shiftLeft( os ) );
}

```

Classes *Person* and *Loginfo* are analogous to the structures in the SunOS example. The only difference being, these structures do not need pointers. Because data can be processed incrementally, it is not necessary to build up linked lists.

```

// common.cc

InputStream &
Loginfo::shiftRight( InputStream & is )
{
    is >> _line >> _host >> _loginat >> _idletime >> _writable;
    return( is );
}

OutputStream &
Loginfo::shiftLeft( OutputStream & os )
{
    os << _line << _host << _loginat << _idletime << _writable;
    return( os );
}

void Loginfo::inspect( OutputStream & os ) { ... }

InputStream &
Person::shiftRight( InputStream & is )
{
    is >> _name >> _realName >> _office >> _officePhone >> _homePhone
        >> _mailLastReceived >> _mailLastRead >> _loggedin;
    return( is );
}

OutputStream &
Person::shiftLeft( OutputStream & os )
{
    os << _name << _realName << _office << _officePhone << _homePhone
        << _mailLastReceived << _mailLastRead << _loggedin;
    return( os );
}

```

```
void Person::inspect( OutputStream & os ) { ... }
```

The overloaded operators << and >> extend the XDR stream model to include the user defined classes *Person* and *Loginfo*. Note the conciseness and elegance of XDR routines in comparison to those in the SunOS example.

```
// client.cc:

int
main( int argc, char * argv[] )
{
    char * host = /* parse argv */;
    char * user = /* parse argv */;
    BufferByteSink uSink( MaxName );
    XDROutputStream xos( uSink );

    int procedure = RFingerProcedureNames;
    if ( user != 0 ) {
        procedure = RFingerProcedureName;
        // User responsible for encoding arguments prior rpccall.
        xos << user;
    }

    RPCClient rfinger( host, RPCProgramRFINGER,
        RFingerVersion, procedure );

    // The ByteSource uSource contains the XDR formatted results.
    ByteSource * uSource = rfinger.call( uSink.buffer(), uSink.length() );

    if ( uSource != 0 ) {
        cerr << "RPC call failed.\n" << flush;
        return;
    }

    XDRInputStream xis( uSource );
    Person me;
    Loginfo loginfo;
    int nextLoginfo;
    int nextPerson;

    xis >> nextPerson;
    while ( nextPerson ) {
        xis >> me;
        me.inspect( cout );
        xis >> nextLoginfo;
        while ( nextLoginfo ) {
            xis >> loginfo;
            loginfo.inspect( cout );
            xis >> nextLoginfo;
        }
        xis >> nextPerson;
    }
}
```

In the decoding of the results, notice the stylistic differences between the SunOS example

and our model. Elements of the list are decoded one at a time, processed and then discarded. No memory allocation is required. Also, in our model, direct access to the XDR encoded data in the network packet is provided by the `ByteSource`, which allows us to shift out, or decode, the results directly from the network packet without incurring an additional copy.

```
// server.cc:

int
getpersonbyname( char * name, XDROutputStream & stream ) { ... }

int
getallpersons( XDROutputStream & stream ) { ... }

int
main( int argc, char * argv[] )
{
    RPCServer rfinger( RPCProgramRFINGER, RFingerVersion, IPProtocolUDP );
    char user[ MaxName ];
    int i;
    Person who;
    Person list[ MaxUser ];

    while ( 1 ) {
        rfinger.getCallMessage();
        if ( rfinger.checkCall() ) {
            switch ( rfinger.procedure() ) {
                case RFingerProcedureNull:
                    rfinger.setAcceptSuccess();
                    break;
                case RFingerProcedureName:
                    rfinger.xMessage() >> user;
                    rfinger.setAcceptSuccess();
                    i = getpersonbyname( user, rfinger.xReply() );
                    if ( i == 0 ) {
                        rfinger.setAcceptGarbageArgs();
                    }
                    break;
                case RFingerProcedureNames:
                    rfinger.setAcceptSuccess();
                    getpersonbyname( rfinger.xReply() );
                    break;
                default:
                    rfinger.setAcceptProcUnavail();
            }
            rfinger.sendReply();
        }
    }
}
```

The routines `getpersonbyname` and `getallpersons` are analogous to the routines found in the SunOS model are responsible for obtaining all the necessary data for the result.

However, by shifting directly onto the `XDROutputStream` supplied by the `RPCServer` object, this function can be written without any temporary memory needed. Also, the `XDROutputStream` is setup to write directly into the outgoing network packet, saving a copy.

## 5 Results

In order to compare the performance of our new stream based XDR/RPC implementation to the SunOS implementation we used the client and server functions described in the previous sections and measured round-trip call times for all the possible combinations of clients and servers. These results are summarized in the table below.

	SunOS Server	New Model Server
SunOS Client	150ms	150ms
New Model Client	160ms	170ms

The numbers indicate that Sun's implementation might be a slightly faster. However, our implementation is not a production model and has not benefited from the amount of man hours put into Sun's implementation. Note, however, the times are very comparable.

Another interesting comparison is the number of lines of code which were necessary to implement both the client and server XDR routines. It took 45 lines to implement the necessary XDR routines for our sample program with the SunOS model while with our new stream model it took only 26 lines of code.

## 6 Availability

The source for the new I/O stream library and the XDR/RPC classes discussed in this paper is public available for free distribution. Interested parties should contact the authors for more information.

## 7 Conclusion

This paper reported on our experiences recasting the Sun Microsystems XDR/RPC protocols in terms of the C++ model of I/O streams. We feel the I/O model resulting from the synthesis is superior to existing XDR/RPC implementations for the following reasons:

- As it turned out many of the problems with the C procedural based I/O model using printf/scanf also existed in the SunOS XDR implementation: type-unsafeness, lack of extensibility, and inflexibility. The C++ stream model removed these problems for normal I/O, and by recasting the XDR protocol into the C++ streams the similar problems found in the procedure based model were removed.
- The idea taken from C++ stream model of separating translation and destination, provides greater accessibility, allowing user defined byte devices to be associated with streams.
- The type of the argument in the stream determines which method to call. This implies the user does not have to specify the routine to be called, leading to more concise and legible code. Further, if the type of the argument changes the correct routine is called.
- Because our RPC call routine only deals with with XDR formatted data, potentially unnecessary copies can be avoided.
- Unnecessary encode and decode routines can be avoided.

- Our RPC model has eliminated many of the memory allocation problems of the Sun's implementation.

Finally, it could be argued that the XDR protocol could have been implemented by a parallel class hierarchy modeled after C++ stream model and not integrated with C++ streams. Such an implementation of XDR would have all the benefits of our model without requiring any changes be made to the C++ stream library or requiring a new implementation of the C++ stream library. We feel such a solution has the drawback of requiring a user to learn two class hierarchies and, even if they are very similar, over time the hierarchies will likely begin to diverge.

## References

- [BN84] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.
- [Sho89] J. E. Shopiro. An Example of Multiple Inheritance in C++: A Model of the Iostream Library. *ACM SIGPLAN Notices*, 25(12):32–36, December 1989.
- [Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
- [Sun85a] Sun Microsystems. *Networking on the SUN Workstation*, 1985.
- [Sun85b] Sun Microsystems. *rpcgen Programming Guide*, 1985.
- [Sun88a] Sun Microsystems. *External Data Representation Standard: Protocol Specification*, 1988.
- [Sun88b] Sun Microsystems. *Network File System: Version 2 Protocol Specification*, 1988.





# Run-Time Type Identification for C++ (Revised)

*Bjarne Stroustrup*

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

*Dmitry Lenkov*

HP Language Labs

## ABSTRACT

This paper describes a proposal for a mechanism for run-time type identification and checked type casts. The mechanism is simple to use, easy to implement, and extensible. This proposal evolved through a series of earlier proposals and ideas. The basic parts of the proposal are a run-time checked type conversion operator (*?type-name*) and an operator `typeid()` that returns objects of class `Type_info` providing a run-time representation of types. Experimental implementations exist. Warning: This is a proposal and the features described may never be accepted into C++.

## 1 Introduction

Consider:

```
class dialog_box : public window {
    // ...
public:
    virtual int ask();
    // ...
};

class dbox_w_str : public dialog_box {
    // ...
public:
    int ask();
    virtual char* get_string();
    // ...
};
```

We may call `ask()` for every `dialog_box` but may call `get_string()` only for `dialog_box`s known to be `dbox_w_str`s. Given only a `dialog_box*` how can we figure out if it really points to a `dbox_w_str`?

There are several ways of defining `dialog_box` and `dbox_w_str` so that the answer can be found. The most popular are to place a type field in `dialog_box` and/or define a virtual function in `dialog_box` that gives the answer. Many C++ libraries provide mechanisms for explicit use of run-time type identification (RTTI) for their classes [3,4,6, and 12] and detailed explanations of how to implement them can be found in [1,5,9,10]. However, these mechanisms are mutually incompatible so that they become a barrier to the use of more than one library. Also, all require a considerable amount of foresight on the part of a base class designer. What is proposed here is a language supported mechanism.

A naive solution would be:

```

void my_fct(dialog_box* bp)
{
    if (typeid(*bp) == typeid(dbox_w_str)) { // is *bp a dbox_w_string?
        dbox_w_str* dbp = (dbox_w_str*)bp;

        // here we can use dbox_w_str::get_string()
    }
    else {

        // 'plain' dialog box
    }
}

```

Given the name of a type as the operand, `typeid()` operator returns an object that identifies it. Given an expression operand, `typeid()` returns an object that identifies the type of the object that the expression denotes. In particular, `typeid(*bp)` returns an object that allows the programmer to ask questions about the type of the object pointed to by `bp`. In this case, we asked if that type was identical to the type `dbox_w_str`.

This is the simplest question to ask, but it is typically *not* the right question. The reason to ask is to see if some detail of a derived class can be safely used. To use it, we need to obtain a pointer to the derived class. In the example, we used a cast on the line following the test. Typically, we are not interested in the *exact* type of the object pointed to, but only in whether we can perform that cast. This question can be asked directly:

```

void my_fct(dialog_box* bp)
{
    dbox_w_str* dbp = (?dbox_w_str*)bp; // checked cast

    if (dbp) {

        // here we can use dbox_w_str::get_string()
    }
    else {

        // 'plain' dialog box
    }
}

```

The checked cast operator `(?T*)p` converts its operand `p` to the desired type `T*` if `*p` really is a `T`; otherwise, the value of `(?T*)p` is 0.

Such a cast is often called *safe* because the result of an attempt to cast a pointer to a wrong type results in the well-defined pointer 0. It is also often called a *downcast* because many people draw class diagrams with derived classes below their bases. To avoid making users overconfident, we prefer to call such casts *checked* rather than *safe*.

Naturally, an implementation of the checked cast will rely on the same kind of information as the `typeid()` operator and share a large part of its implementation.

There are several advantages to merging the test and the cast into a single checked cast operation:

- The checked cast notation is less verbose than alternatives using named operations.
- By using the information available in the type information objects it is often possible to cast from a virtual base class to a derived class; see Appendix B.
- By using the information available in the type information objects it is possible to cast to types that are not fully defined in the scope of the cast; see §8.
- A checked cast makes it impossible to mismatch the test and the cast.

As examples of such mismatches, consider:

```

void my_fct(dialog_box* bp)
{
    if (typeid(*bp) == typeid(dialog_box)) { // check, then cast
        dbbox_w_str* dbp = (dbbox_w_str*)bp;

        // here we can use dbbox_w_str::get_string()
    }

    // ...
}

```

where the user checked against the type of the base class `dialog_box` instead of the derived class `dbbox_w_str`, and

```

void my_fct(dialog_box* bp)
{
    if (typeid(*bp) != typeid(dbbox_w_str)) { // check, then cast
        dbbox_w_str* dbp = (dbbox_w_str*)bp;

        // here we can use dbbox_w_str::get_string()
    }

    // ...
}

```

where the user applied the explicit cast on the wrong branch of the `if` statement. Both kinds of errors have been seen in real systems.

The notation is still redundant in that `dbbox_w_str` is mentioned twice in

```

dbbox_w_str* dbp = (?dbbox_w_str*) bp;

```

However, removing that redundancy would leave the programmer without a clearly visible clue that something “interesting” is going on. This redundancy also enables an added degree of checking:

```

extern void f(dbbox_w_str* dbp);

// ...

void g(dialog_box* bp)
{
    f(bp); // error: cannot (implicitly) convert
           // from a base to a derived class

    f((?dbbox_w_str*)bp); // ok: checked cast
}

```

The *(?type-name)* notation was chosen to parallel the traditional *(type-name)* cast notation. It has the advantage over the traditional notation that it is easy to spot in a program – both for a human and for a simple search tool (for example, `grep`).

As a final simplification we might adopt the Algol68 notion that declarations yield values and thereby allow declarations in conditions. We could then write this:

```

void my_fct(dialog_box* bp)
{
    if (dbbox_w_str* dbp = (?dbbox_w_str*) bp) {

        // use 'dbp'
    }

    // ...
}

```

The value of a declaration is the value of the declared variable after initialization. To avoid ambiguities, we do not suggest that declarations should be allowed in any new places in the grammar except as conditions.

See Appendix A for further details.

In §8 we will return to the `typeid()` operator and examine what it and the objects it returns are good for.

## 2 Uses and Misuses of RTTI

One should use explicit run-time type information only when one has to; static (compile-time) checking is safer, implies less overhead, and – where applicable – leads to better structured programs. For example, RTTI can be used to write thinly disguised switch statements:

```
// misuse of run-time type information:

void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
        // do nothing
    }
    else if (typeid(r) == typeid(Triangle)) {
        // rotate triangle
    }
    else if (typeid(r) == typeid(Square)) {
        // rotate square
    }
    // ...
}
```

This style of code is usually best avoided through the use of virtual functions. It was the first author's experience with Simula code written this way that caused facilities for run-time type identification to be left out of C++ in the first place.

For many people trained in languages such as C, Pascal, Modula, Ada, etc. there is an almost irresistible urge to organize software as a set of switch statements. This urge should most often be resisted. Please note that even though we are proposing a RTTI mechanism for C++ we do not propose to support it with a type-switch statement (such as Simula's INSPECT statement, for example).

Many examples of proper use of RTTI arise where some service code is expressed in terms of one class and a user wants to add functionality through derivation. The `dialog_box` example from §1 is an example of this. If the user is willing and able to modify the definitions of the library classes, say `dialog_box`, then the use of RTTI can be avoided; if not, it is needed. Even if the user is willing to modify the base classes, such modification may have its own problems. For example, it may be necessary to introduce dummy implementations of virtual functions such as `get_string()` in classes for which the virtual functions are not needed or not meaningful.

For people with a background in languages that rely heavily on dynamic type checking, such as Smalltalk, it is tempting to RTTI and overly general types. For example:

```
// misuse of run-time type information:

class Object { /* ... */ };

class Container : public Object {
    // ...
public:
    void put(Object*);
    Object* get();
    // ...
};
```

```

class Ship : public Object { /* ... */ };

Ship* f(Ship* p1, Container* c)
{
    c->put(p1);
    // ...
    Object* p2 = c->get();
    if (Ship* p3 = (?Ship*) p2) // run-time type check
        return p3;
    else {
        // do something else
    }
}

```

Here, class Object is an unnecessary implementation artifact. Problems of this kind are often better solved by using container templates holding only a single kind of pointer:

```

template<class T> class Container {
    // ...
public:
    void put(T*);
    T* get();
    // ...
};

Ship* f(Ship* p1, Container<Ship>* c)
{
    c->put(p1);
    // ...
    return c->get();
}

```

Combined with the use of virtual functions, this technique handles most cases.

RTTI can be a reasonable choice where the type of an object returned from some function cannot be determined at compile time from the types of its arguments. For example, consider a couple of classes where objects can be compared using information from a common base class only:

```

class X {
    // ...
public:
    X* greater(X* arg); // return greater of *this and *arg
    // ...
};

class D1 : public X { /* ... */ };
class D2 : public X { /* ... */ };

void f(D1* a, D2* b)
{
    X* res = a->greater(b);
    if (D1* p = (?D1*)res) {
        // ...
    }
    else {
        // ...
    }
}

```

Note that there is no requirement that objects should only be compared to objects of their own type, and that the type of the returned object cannot be determined from the types of the operands only. Had either of those conditions been true, superior solutions could have been achieved without using RTTI. The recent relaxation of the virtual function overriding rules also provides an alternative to RTTI in some cases; see

## Appendix E.

Finally, RTTI has an important role in optimizations. Consider a function using an abstract set class:

```
void fct(set<T>* s)
{
    for (T* p = s->first(); p; p = s->next()) {

        // ordinary set algorithm
    }

    // ...
}
```

This is nice and general, but what if we knew that many of the sets passed were implemented by singly linked lists, `slists`, if we knew an algorithm for the loop that was significantly more efficient for lists than for general sets, and if we knew (from measurement) that this loop was a bottleneck for our system? It would then be worth our while to expand our code to handle `slists` separately:

```
void fct(set<T>* s)
{
    if (slist<T>* sl = (?slist<T>*)s) { // s is an slist

        for (T* p = sl->first(); p; p = sl->next()) {

            // souped up list algorithm
        }
    }
    else {
        for (T* p = s->first(); p; p = s->next()) {

            // ordinary set algorithm
        }
    }

    // ...
}
```

Naturally, this leads to messier code and makes `fct()` depend directly on the `slist` class, but that can sometimes be a worthwhile price to pay. In particular, in the case above we not only get the benefit from an improved `slist` algorithm but also avoid virtual function calls (on the abstract class `set`) in favor of inline functions (on the concrete class `slist`). Combined, these two optimizations can amount to one or two orders of magnitude. Please note that the “optimized” example is still as general as the original. It handles every argument properly (as opposed to the buggy `Shape` example above). All that has been done is to insert code dealing with an important special case. Should the representations used for `set<T>`s change, the grimy optimization code will simply become redundant; it will not become a source of bugs caused by false assumptions.

Evidence from library design and use suggests that almost everybody needs RTTI occasionally, but that one should aim to design systems so as to minimize its use. Where applicable, static type checking provides stronger guarantees, smaller and faster code, and cleaner designs. Therefore RTTI should only be used where it is clearly needed. One should be suspicious of “arguments” of the form “RTTI is clearly needed in this case.” In our experience, such arguments are often wrong and hide a lack of understanding of the problem area or of the design choices available in C++.

### 3 Checked and Unchecked Casts

The introduction of run-time type identification separates objects into two categories: The ones that have run-time type information associated so that their type can be determined (almost) independently of context and those that haven't. Why? We cannot impose the burden of being able to identify an object's type at run-time on built-in types such as `int` and `double` without unacceptable costs in run-time, space, and layout compatibility problems. A similar argument applies to simple class objects and C-style structs.

Consequently, from an implementation point of view, the first acceptable dividing line is between objects of classes with virtual functions and classes without. The former can easily provide run-time type information, the latter cannot.

Further, a class with virtual functions is often called a polymorphic class and polymorphic classes are the only ones that can be safely manipulated through a base class<sup>†</sup>. It thus, from a programming point of view, seems natural to provide run-time type identification for polymorphic types (only): They are exactly the ones for which C++ supports manipulation through a base class. Supporting RTTI for a non-polymorphic type would simply provide support for switch-on-type-field programming. Naturally the language should not make that style impossible, but we see no need to complicate the language solely to accommodate it.

Experience shows that providing RTTI for polymorphic types (only) works acceptably. However, people can get confused about which objects are polymorphic and thus about whether a checked cast can be used. This is discussed further in Appendix C.

Applying the checked cast (`?T*`) to a pointer `p` of a non-polymorphic type is a compile time error. Given checked casts, an ordinary cast from a polymorphic type could be considered suspicious and we expect that good compilers will optionally issue warnings for such casts. For example:

```
class X {
    // no virtual functions
};

class B {
    virtual int f();
    // ...
};

void f(X* px, B* pb)
{
    Y* p = (?Y*)px; // error: X is not polymorphic
    D* q = (D*)pb;  // optional warning: B is polymorphic,
                   // you could have used (?D*)
}
```

Note that checked and unchecked casts are fundamentally different in that an unchecked cast is based (almost) exclusively on type information whereas a checked cast is based (almost) exclusively on the value of the object.

A checked cast of pointer with the value 0 yields 0 because 0 does not point to an object of a polymorphic type. For example:

```
X* p = 0;

Y* q1 = (?Y)p; // q1 = 0
Y* q1 = (?Y)0; // compile time error
```

The relationship between checked and unchecked casts is discussed further in §8. Checked casts of references are considered in §5. Syntax issues are discussed in §8.

#### 4 Cross Hierarchy Casting

Two related questions must be answered:

- Should casting be constrained to derivation relationships known at compile time?
- Should it be possible to cast from a class to a sibling class in a multiple inheritance hierarchy?

For example:

---

<sup>†</sup> Here “safely” means that the language provides guarantees that objects are used only according to their defined type. Naturally, individual programmers can in specific cases demonstrate that manipulations of a non-polymorphic don’t violate the type system.

```

class A { /* ... */ virtual void f(); };
class B { /* ... */ virtual void g(); };
class D : public A, public B { /* ... */ };
class X;

void f(A* pa)
{
    X* px = (?X*)pa; // X undefined: legal?
    B* pb = (?B*)pa; // B apparently unrelated to A: legal?
}

```

In both cases checking is possible and performing it is useful, thus both cases are legal.

In the case of a checked cast to an undefined class this decision ensures that the same result is obtained independently of whether the class declaration has been seen or not. This is not the case for ordinary casts; see §8. Note that a checked cast requires its operand to be of a known and polymorphic type.

Consider the following set of classes:

```

class employee { /* ... */ };
class manager : public employee { /* ... */ };
class analyst : public employee { /* ... */ };

class engineer { /* ... */ };
class electrical_engineer : public engineer { /* ... */ };
class mechanical_engineer : public engineer { /* ... */ };

```

If we want to ask questions like:

- Is this engineer a manager ?
- Does this employee have an EE degree ?
- How many analysts have an engineering degree ?

and we want to use language features rather than algorithms based on data stored by the programmer, then we define:

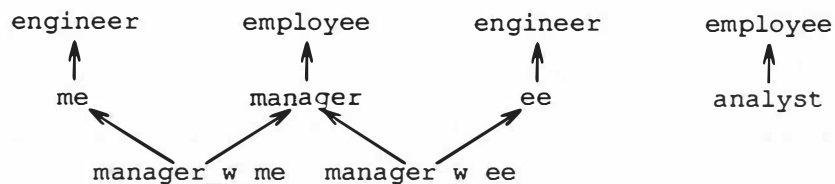
```

class manager_with_ee
    : public manager, public electrical_engineer
{ /* ... */ };

class manager_with_me
    : public manager, public mechanical_engineer
{ /* ... */ };

```

Or graphically:



We can then use checked casts like this:



```

my_fct(engineer* pe1, employee* pe2)
{
    if (manager* m = (?manager*)pe1) {
        // this engineer is a manager
    }
    // ...
    if (electrical_engineer* ee = (?electrical_engineer*)pe2) {
        // this employee has an EE degree
    }
    // ...
}

```

Note that we can do this even where the connection between `employee` and `electrical_engineer` is unknown because no class derived from both, such as `manager_with_ee`, has yet been defined. In general, it is not possible to know that two classes are unrelated because there is always the possibility that a class defined in some other compilation unit is derived from both. However, given an object of polymorphic type we can always (at run time) determine if the classes are related for that object.

The decision to allow cross-hierarchy casting also matches the rule that a virtual function can be defined on one branch of a multiple inheritance hierarchy and called through another.

## 5 References

The discussion thus far has focussed on pointers. However, a reference can also refer to objects of a variety of base and derived classes and is subject to casting in a way very similar to pointers. For example, the set example from §2 could be written using references instead of pointers. However, we cannot simply rewrite the critical test

```
if (slist<T>* sl = (?slist<T>*)s)
```

to

```
if (slist<T>& sl = (?slist<T>&)s)
```

That wouldn't make sense in general because there is no "zero reference" to test. Consequently, a reference cast throws an exception if the cast cannot be performed. The example thus becomes:

```

void my(set<T>& s)
{
    try {
        slist<T>& sl = (?slist<T>&)s; // s is an slist

        for (T* p = sl.first(); p; p = sl.next()) {

            // souped up list algorithm
        }
    }
    catch(Bad_cast) {
        for (T* p = s.first(); p; p = s.next()) {

            // ordinary set algorithm
        }
    }
    // ...
}

```

This is a very poor example of a reference cast because it uses an exception to handle ordinary local control flow rather than an error. In this case, a pointer cast would have been more appropriate:

```

    if (slist<T>* p = (?slist<T>*)&s) { // s is an slist
        slist<T>& sl = *p;
        // ...
    }

```

The difference in results of a failed checked pointer cast and a failed checked reference cast reflects a fundamental difference between references and pointers. A pointer may or may not point to an object, whereas a reference may be assumed to refer to one. As ever, the possibility of zero pointers makes explicit tests necessary where pointers are used.

Explicit tests against 0 can be – and therefore occasionally will be – accidentally omitted. One might argue that a checked pointer cast that fails should throw an exception just like a failed checked reference cast. However, this would only handle one minor source of 0 pointers and not all 0 pointers lead to errors. The programmer has a choice, though:

```

void f(dialog_box* p)
{
    dbx_w_string* p1 = (?dbx_w_string*)p; // p or 0
    dbx_w_string* p2 = (?dbx_w_string*)&p; // p or exception
    dbx_w_string* p3 = &(?dbx_w_string&)*p; // p or exception
    // ...
}

```

A checked reference cast can be a good way of testing an assumption. In contrast, the checked pointer cast allows (and requires) a test to select between two reasonable alternatives.

## 6 How Much Information?

The basic notion of the RTTI mechanisms described here is that for maximal ease of programming and implementation independence we should minimize the use of RTTI:

- [1] Preferably, we should use no run-time type information at all and rely exclusively on static (compile time) checking.
- [2] If that is not possible, we should use only checked casts. In that case, we don't even have to know the exact name of the object's type and don't need to include any header files related to RTTI.
- [3] If we must, we can compare `typeid`s, but to do that we need to know the exact name of at least some of the types involved. It is assumed that "ordinary users" will never need to examine run-time type information further.
- [4] Finally, if we absolutely do need more information about a type – say because we are trying to implement a debugger, a data base system, or some other form of object I/O system [1] – we can use operations on `typeid`s to obtain more detailed information.

This approach of providing a series of facilities of increasing involvement with run-time properties of classes contrasts to the approach of providing a class giving a single standard view of the run-time type properties of classes. We feel that the proposed approach encourages greater reliance of the (more safer and efficient) static type system, has a smaller minimal cost (in time and comprehensibility) to users, and is also more general because of the possibility of providing multiple views of a class by providing more detailed type information.

### The `typeid()` Operator

In §1, we presented the `typeid()` operator only briefly before making its use implicit in the checked cast mechanism. However, `typeid()` can be used explicitly to gain access to information about types at run time; `typeid()` is a built-in operator. Had it been a function its declaration would have looked something like this:

```

class Type_info;
const Type_info& typeid(type-name); // pseudo declaration
const Type_info& typeid(expression); // pseudo declaration

```

That is, `typeid()` returns a reference to an unknown type called `Type_info`. Given a *type-name* as its operand, `typeid()` returns a reference to a `Type_info` that represents the *type-name*. Given an

*expression* as its operand, `typeid()` returns a reference to a `Type_info` that represents the type of the object denoted by the *expression*. For example:

```
class B { /* ... */ virtual void f(); }; // a polymorphic base class
class D : public B { /* ... */ };

B* p = new D; // a B* pointing to a D
B& r = *p;    // a B& referring to a D
int i;

typeid(p) == typeid(B*)
typeid(*p) == typeid(D)
typeid(r) == typeid(D)
typeid(&r) == typeid(B*)
typeid(7) == typeid(int)
typeid(0) == typeid(int)
typeid(i) == typeid(int)
typeid(&i) == typeid(int*)
```

Note that for a polymorphic type accessed through a pointer or a reference the actual object is examined and its (dynamic) type returned. For a non-polymorphic type the object returned represents the static type:

```
class X { int i }; // a non-polymorphic class
class Y : public X { int j; };

X* xp = new Y; // unwise: typeid(*xp) == typeid(X)
```

As ever, manipulating a non-polymorphic class through a base class relies on the programmer knowing exactly what is being done.

Because `typeid(*p)` involves examining the object `*p` the case `p==0` presents a problem. The solution is to throw an exception:

```
p = 0;
typeid(*p); // throw Bad_typeid
```

Naturally, a simple test prevents the exception:

```
if (p == 0) {
    // ...
}
else {
    typeid(*p);
    // ...
}
```

Actually, `typeid()` slightly favors the use of references:

```
void f(B& r)
{
    if (typeid(r) == typeid(D) {
        // use r as a D
    }

    // used r as a plain B
}
```

Here we are entitled to assume that `r` refers to an object and we don't have to decorate `r` with any operators the way we had to decorate a pointer `p` with a dereference operator to get the type of the object `*p`.

The reason `typeid()` returns a reference to `Type_info` rather than a pointer is that it is not clear that every implementation will be able to guarantee uniqueness of type identification objects. In particular, it is not obvious that every dynamic loading and linking mechanism will be able to avoid occasional duplication of such objects. With a `Type_info&` there is no problem defining `==` to cope with such duplication.

Some `typeid()`s can be obtained only using the `typeid(typeName)` syntax. For example:

```
char& r = obj;
typeid(r) == typeid(char) // NOT typeid(char&)
```

It is possible, however, to express the `typeid()` for every type that an object can have. This is important for writing code, such as some object I/O systems, that relies on using descriptions of objects at run-time.

### Class `Type_info`

Class `Type_info` is defined in the standard header file `<Type_info.h>` which needs to be included for the result of `typeid()` to be used. The exact definition of class `Type_info` is implementation dependent, but it is a polymorphic type that supplies comparisons and an operation that returns the name of the type represented:

```
class Type_info {
    // implementation dependent representation
private:
    Type_info(const Type_info&);           // objects cannot
    Type_info& operator=(const Type_info&); // be copied by users
public:
    virtual ~Type_info();                 // is polymorphic

    int operator==(const Type_info&) const; // can be compared
    int operator!=(const Type_info&) const;

    const char* name() const;             // get the type name
};
```

More detailed information can be supplied and accessed as described below. However, because of the great diversity of the "more detailed information" desired by different people and because of the desire for minimal space overhead by others, the services offered by `Type_info` are deliberately minimal.

### Extended Type Information

Consider how an implementation or a tool could make information about types available to users at run-time. Say we have a tool that generates a table of (*member\_name, offset, typeid*) entries for each member of a class. The preferred way of presenting this to the user is to provide an associative array (map, dictionary) of type names and such tables. To get such a member table for a type a user would write:

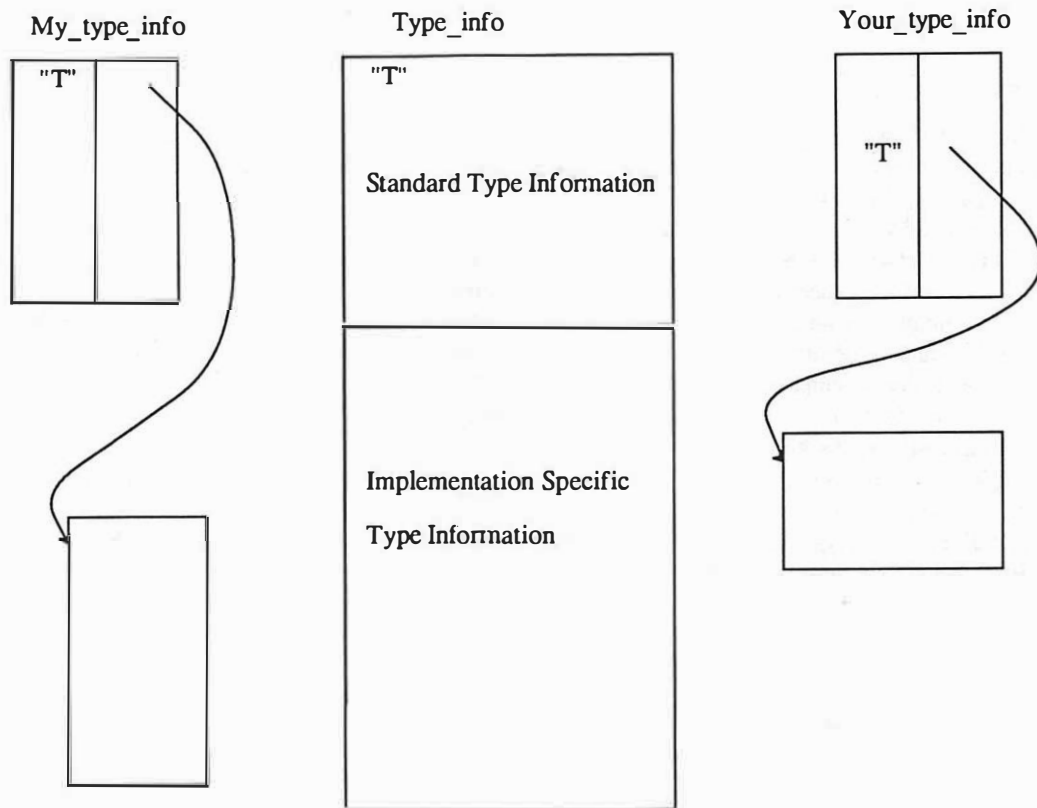
```
void f(B* p)
{
    My_member_info* pi = my_type_table[typeid(p).name()];
    // use *pi
}
```

where `My_member_info` is the name of the type of our information, and `my_type_table` is the name of the associative array in which we keep the (*typename, My\_member\_info\**) pairs. If we wanted to, we could index the tables directly with `typeids` rather than requiring the user to use the `name()` string:

```
My_member_info* pi = my_type_table[typeid(p)];
```

It is important to note that this way of associating `typeids` with information allows several people or tools to associate different information to types without interfering with each other. This is most important because the likelihood that someone can come up with a set of information that satisfies all users is zero. In particular, any set of information that would satisfy most users would be so large that it would be unacceptable overhead for users that need only minimal run-time type information.

Using these techniques, we might have several independent sets of information about types in a program:



The function `typeid::name()` is logically redundant in that the name string could be obtained through the association technique described above. However, that wouldn't allow association tables to be sorted according to the spelling of type names and would make it less easy for programmers to obtain string representations of type names. We would prefer it to be trivially easy to print the name of a class. For example:

```
#include <Type_info.h>

template<class T> class Vector {
    // ...
    void my_name1() { cout << "Vector<" << typeid(T).name() << '>'; }
    void my_name2() { cout << typeid(Vector<T>).name(); }
    void my_name3() { cout << typeid(Vector).name(); }
};
```

where all functions happen to be equivalent.

What information might a tool or an implementation make available to a user? Basically any information that a compiler can provide and that some program might want to take advantage of at run time. For example:

- Object layouts for object I/O and/or debugging.
- Tables of functions together with their symbolic names for calls from interpreter code.
- Lists of all objects of a given type.
- References to source code for the member function.
- Online documentation for the class.

The reason such things are supported through libraries, possibly standard libraries, is that there are too many needs, too many potentially implementation specific details, and too much information to support every use in the language itself. Also, some of these uses subvert the static checking provided by the

language. Others impose costs in run time and space that we do not feel appropriate for a language feature.

## 7 Implementation Issues

Consider how to implement RTTI. The `typeid()` operator and the checked cast notation `(?T)` affects syntax checking and type checking minimally. To deal with run-time aspects of the mechanism three separate issues must be addressed:

- [1] How do we get hold of run-time type information given a pointer or a reference?
- [2] How do we use the run-time type information to implement `typeid()` and checked casts?
- [3] How do we generate the run-time type information?

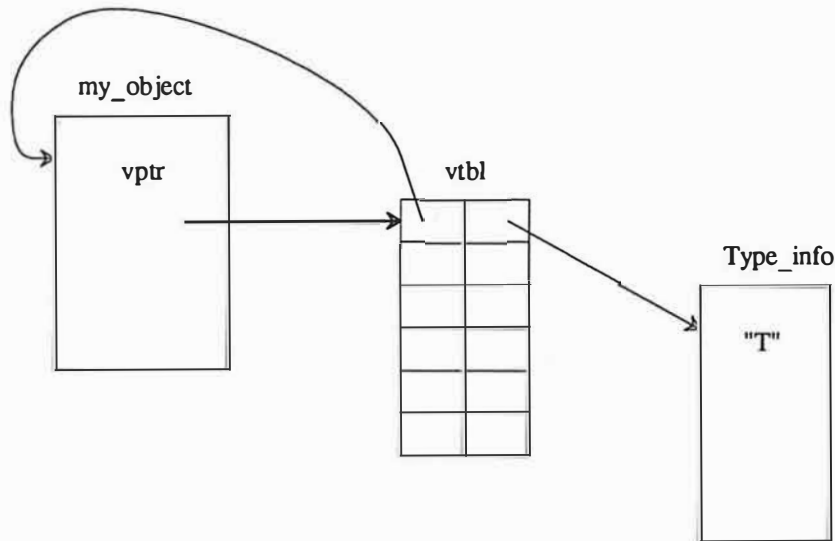
The implementation described here is only one of several possible. It assumes a traditional and fairly straightforward implementation of C++ along the lines described in [2]. That is, each object of a class with virtual functions contains a pointer (`vptr`) to a table of virtual functions (`vtbl`).

The basic idea is to place a pointer to an object describing an object's type in the `vtbl`. Such description objects will be of some type derived from class `Type_info`.

Basically `typeid(expression)` is nothing but a test to protect against zero-valued pointers followed by a double indirection to retrieve the pointer to the `Type_info` object.

A call `typeid(type-name)` degenerates into the name of the type's `Type_info` object.

Here is a plausible memory layout for an object with virtual function table and type information object:



For each type with virtual functions an object of type `Type_info` is generated. These objects need not be unique. However, a good implementation will generate unique `Type_info` objects wherever possible and only generate `Type_info` objects for types where some form of run-time type information is actually used. An easy implementation simply places the `Type_info` object for a class right next to its `vtbl`.

### Checked Casts

In most cases the implementation of a cast `(?D*)px` where the static type of `*px` is `X` is straightforward: retrieve a pointer to the run-time type identification object from `*px`, generate a pointer to the run-time type identification object for `D`, and have a library routine see if `*px`'s class is `D` or a base of `D` and return a – possibly slightly adjusted – pointer. The adjustment is needed when `X` class isn't a first base of `D` class. For example:

```

class D : public A, public X { /* ... */ };

void f()
{
    X* px = new D;    // px doesn't point to the start of the D object
    D* pd = (?D*)px;  // pd should point to the start of the D object
}

```

This adjustment is trivially implemented.

However, cases where a base class X appears more than once in a class hierarchy need more care. Consider first ordinary (non-virtual) base classes:

```

class D1 : public X { /* ... */ };
class D2 : public X { /* ... */ };
class D : public D1, public D2 { /* ... */ };

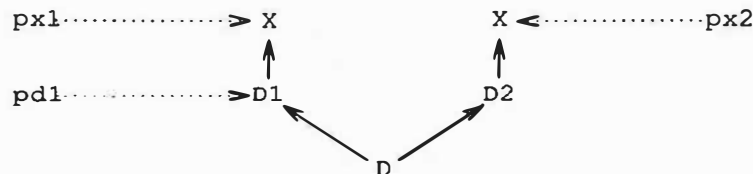
void f(D* pd)
{
    X* px1 = (D1*)pd;
    X* px2 = (D2*)pd;

    pd = (?D*)px1; // pd should point to the start of the D object
    pd = (?D*)px2; // pd should point to the start of the D object

    D1* pd1 = (?D1*)px1;
    pd1 = (?D1*)px2;
}

```

Or graphically



Clearly the adjustments needed for the two `(?D*)` casts are different. Similarly, the adjustments needed for the two `(?D1*)` casts are different. Consequently, we need to store (in the `vtbl` or equivalent) the offset of the sub-object in the overall object. Given that, we can not only perform the correct adjustment of pointers but also resolve the case of multiple sub-objects. Virtual base classes are handled slightly differently; see Appendix B.

## 8 Alternatives

The current proposal is a result of a series of ideas and experiments with both the syntax and semantics of run-time type identification. Here, we would like to explain some of the alternatives we considered. The ideals we looked for were the usual: Ease of learning, ease of reading, direct representation of the underlying semantics, no pointless redundancy, minimal syntactic innovation, minimal compatibility problems (including a minimal number of new keywords), ease of implementation, reasonable run-time and space efficiency, etc.

### Checked and Ordinary Casts

Cast is one of the most error-prone facilities in C++. It is also one of the ugliest syntactically. Naturally we considered if we could

- [1] eliminate casts, and if not then
- [2] make casts safe, and if not then at least
- [3] provide a cast syntax that makes it obvious that an unsafe operation is used.

Basically, this proposal reflects our conclusion that none of the above are feasible in C++ at this time so we

must settle on the policy that has been followed for years:

- [4] Provide alternatives to casting and discourage the use of casts.

Considering [1], we observed that no language supporting systems programming has completely eliminated the possibility of casting and that even effective support for numeric work requires some form of type conversion. Thus the aim must be to minimize the use of casts and make them as well behaved as possible. Starting from that premise we devised a proposal that unified checked and ordinary casts using a single syntax [11]. This seemed a good idea, but upon closer examination several problems were uncovered:

- [1] Checked casts and ordinary unchecked casts are fundamentally different operations. Checked casts look into objects to produce a result and may fail giving a run-time indication of that failure. Ordinary casts perform an operation that is determined exclusively by the types involved and doesn't depend on the value of the object involved (except for occasional checking for 0 pointers). An ordinary cast doesn't fail; it simply produces a new value. Using the cast syntax for both checked and unchecked casts led to confusion about what a give cast expression really did.
- [2] If checked casts are not syntactically distinguished it is not possible to find them easily (grep for them, to use Unix-speak).
- [3] If checked casts are not syntactically distinguished then it is not possible to have the compiler check for unsuitable uses of checked casts. If distinguished, we can make it an error to attempt a checked cast for objects that don't support run-time checking.
- [4] Programs using ordinary casts would have their meaning changed if run-time checking were applied wherever feasible. Examples are casts to undefined classes and casts within multiple inheritance hierarchies. We did not manage to convince ourselves that this change of meaning would never break a reasonable program.
- [5] The cost of checking would be incurred even for old programs that already carefully checked that casts were viable using other means.
- [6] The suggested way of "turning off checking," casting to and from `void*`, wouldn't be perfectly reliable because the meaning would be changed in some cases. These cases might be perverted, but because understanding of the code would be required the process of "turning off checking" would be manual and error-prone. We are also against techniques that would add yet more uncheckable casts to programs.
- [7] Making some casts "safe" would make casting more respectable; yet the long-term aim is to decrease the use of all casts (including checked casts).

After much discussion we found this formulation: "Would our ideal language have more than one notation for type conversion?" For a language that distinguishes fundamentally different operations syntactically the answer is "yes." Consequently we abandoned the attempt to "hijack" the old cast syntax.

We considered if it would be possible to "deprecate" the old cast syntax in favor of something like:

```
Checked<T*>(p);    // checked conversion of p to a T*
Unchecked<T*>(p);  // unchecked conversion of p to a T*
```

This would eventually make all conversions obvious, thus eliminating the problem that traditional casts are too hard to spot in C and C++ programs. It would also give all casts a common syntactic pattern and share the `<T*>` notation for types with templates. This line of development was abandoned (for now) because we realized that even though `Checked<T*>(p)` looks a bit like a template it cannot in fact be defined as a template. Thus we would have another syntactic oddity on our hands. Most likely, there would also be the traditional uproar over the introduction of new keywords to contend with. Finally, we considered it possible that the inevitable confusion over compatibility and transition issues might derail the consideration of run-time type identification so that we would end up with no improvements at all.

The notion of checked casts extends cleanly to arithmetic types. The meaning of `(?T) v` would be "if the value `v` can be represented as a `T` return that representation; otherwise throw `Bad_cast`." The use of an exception is necessary because many arithmetic types does not have a distinguished value (like a zero-pointer or NaN) that we could return for the user to test for. This facility would be a pure extension to the current proposal, but we decided not to complicate matters by adding it now.



## Implementation and Tool Concerns

A key line of thought was to try to define a notation for run-time type identification that did not involve anything a user couldn't define in C++ itself; that is, trying to guarantee that the new mechanisms would fit smoothly into the language by actually defining them in the language and then relying on compilers and other tools for optimization.

We were only partially successful. Our previous proposal [11] had that property, but providing it involved notations and concepts that many deemed confusing and too complicated.

The proposed solution involves three extensions to the syntax:

- [1] The `(?type-name)` syntax for checked casts.
- [2] The `typeid(type-name)` syntax for "typeid literals."
- [2] The `typeid(expression)` syntax for getting type information from an object.

## The `typeid()` Operator

We felt that the `typeid()` operator was more appropriate than a "magic" member function that could be applied to all objects. Had we defined `typeid()` as a member function we would have had to allow something like:

```
void f(X* p, Y& r, int i, char*a[])
{
    p->typeid();
    r.typeid();
    i.typeid();
    a.typeid();
    X::typeid();
    int::typeid();
    char*::typeid();
}
```

Once all possibilities had been taken into account, the "magic" member function solutions looked messy.

## Type Relations

We considered defining `<`, `<=`, etc., on `Type_info` objects to express relationships in a class hierarchy. That is easy, but too cute. It also suffers from the problems with an explicit type comparison operation as described in §1. We need a cast in any event so we can just as well use a checked cast.

## Unconstrained Methods

There are many ways of using run-time type information in a language and a diverse set of facilities has been used in programming languages. We considered a couple of alternatives with implications beyond run-time type identification. Given RTTI, one can support "unconstrained methods;" that is, one could hold enough information in the RTTI for a class to check at run time whether a given function was supported or not. Thus one could support Smalltalk-style dynamically-checked functions. However, we felt no need for that and considered that extension as contrary to our effort to encourage efficient and type-safe programming. In other words, that extension would take C++ in a new direction contrary to its direction so far. The checked cast enables a check-and-call strategy:

```
if (D* pd = (?D*)pb) { // is *pb a D?
    pd->dfct(); // call D function
    // ...
}
```

rather than the call-and-have-the-call-check strategy of Smalltalk:

```
pb->dfct(); // hope pd has a dfct
```

The check-and-call strategy provides more static checking (we know at compile time that `dfct` is defined for class `D`), doesn't impose an overhead on the vast majority of calls that don't need the check, and provides a visible clue that something beyond the ordinary is going on.

## Multi-methods

A more promising use of RTTI would be to support "multi-methods," that is, the ability to select a virtual function based on more than one object. Such a language facility would be a boon to writers of code that deals with binary operations on diverse objects. Generalized addition, geometric intersect operations, and other reasonably common operations belong to this class of problem. We make no such proposal, however, because we cannot clearly grasp the implications of such a change and do not want to propose a major new extension without experience in C++. In the context of C++, we would have to work out argument conversions and ambiguity rules, find a call mechanism that approached the virtual call mechanism in efficiency, and work out the interaction between multi-method declarations and separate compilation.

## 9 Survey of Issues

There are several issues and proposals wrapped up into the RTTI mechanism. They can and should be considered individually but we feel that the final evaluation of any run-time type identification scheme should be based on the utility and elegance of a complete set of features. The individual aspects of the proposal here are:

- [1] We use checked casts. The alternatives are checking all casts where sufficient information is available (§8) or relying on some alternative notion such as an `isKindOf` operator (§1, §8) or a relational operator on `typeid()`s (§8) for determining inheritance relationships.
- [2] We use virtual functions to distinguish types that support run-time type identification from types that don't. The alternative would be to support RTTI for all types or to support RTTI for types explicitly declared to support it (§3, Appendix C).
- [3] We use a syntax extension to allow declarations in conditions (Appendix A).
- [4] We allow cross hierarchy casting. The alternative is to allow casts only within known class hierarchies (§4).
- [5] We use reference casts. The alternative is to disallow reference casts and thus avoiding the use of exceptions (§5).
- [6] We disallow objects of non-polymorphic types as operands for checked casts. The alternative is to interpret such checked casts as ordinary unchecked casts (§3). In addition, one might support checked casts for arithmetic types also (§8).
- [7] We allow casts to a non-unique sub-object from within an object. The alternative is to define casting as conversion from the run-time determined class of the object to the desired type and then consider a cast to a non-unique sub-object ambiguous (§7).
- [8] We use the `typeid` operator (§6). The alternatives is either to provide no way of getting access to objects describing a type or to provide a complete `typeid` type for manipulating type identifiers instead of using `Type_info` objects directly [11].
- [9] We allow non-polymorphic types as operands for `typeid()` and in such cases `typeid()` yields values that depend on the static type of its operand. The alternatives is to cause compile time errors or supplying RTTI for every object (§6).
- [10] We allow expressions of any type as operands to `typeid()`. The alternative is to accept pointers and/or references only (Appendix D).
- [11] We use a `Type_info` class defined in a standard library. The alternative is to support checked casts and type identity only (§6).
- [12] We use a minimal `Type_info` class. The alternative is to guarantee the presence of a much more extensive type information class.

There are of course many additional details, such as the exact name of the `Bad_cast` and `Bad_typeid` exceptions, but we feel that any RTTI facility designed along the lines we suggest will be characterized by the choices outlined here.

## 10 How to Manage until RTTI comes

This proposal for RTTI is most unlikely to be available on your C++ implementation any day soon. What can you do to get the benefits until some variant RTTI becomes generally available? If you use one of the major libraries, you already have some mechanism available and even if you don't you can build your own using the technique described in [10]. The real problem is how to stay compatible with others

and to make sure that you can convert the “real” RTTI system once it becomes available.

We suggest you write your code in terms of five macros

```
const Type_info& static_type_info(type)    // get Type_info for type
const Type_info& ptr_type_info(pointer)    // get Type_info for pointer
const Type_info& ref_type_info(reference)  // get Type_info for reference
pointer ptr_cast (type, pointer)           // convert pointer to type*
reference ref_cast (type, reference)       // convert reference to type&
```

We believe that these can be defined for any reasonable RTTI mechanism so that your user code becomes independent of its particulars. That makes portability manageable and once your C++ implementation provides a standard RTTI mechanism you can either redefine your macros or (preferably) rewrite the code to use it directly.

## 11 Acknowledgements

Jim Coplien, Brian Kernighan, Andrew Koenig, Doug McIlroy, Rob Murray, and Jonathan Shopiro provided valuable insights that helped shape this proposal. Tom Penello checked that allowing declarations in conditions would not introduce any new syntax ambiguities. Michey Mehta and Shankar Unni provided many ideas of different approaches to run-time type identification and its implementation that helped better understand problems and solutions presented in this proposal. Steve Clamage found (too) many minor mistakes in an earlier version of this paper.

The current proposal evolved from the one presented to the ANSI/ISO C++ committee for discussion and published to solicit further comments [11]. The discussion extensions working group ANSI/ISO C++ committee at the London meeting was particularly useful. We found almost universal application of run-time type identification in various forms, confirmed the general structure of the proposal, and – somewhat to our surprise – demonstrated that ordinary and checked casts could not be unified by a single syntax. Thanks to all who took part in that discussion.

## 12 References

- [1] Frank Buschmann, Konrad Kiefer, and Michael Stal: *A Runtime Information System for C++*. Proc. TOOLS Europe 1992.
- [2] Margaret A. Ellis, Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
- [3] Mary Fontana, Martin Neath: *Checked Out And Long Overdue: Experience in the Design of a C++ Class Library*. USENIX C++ Conference Proceedings, April, 1991.
- [4] Keith E. Gorlen: *An Object-Oriented Class Library for C++ Programs*. Proceedings of the USENIX C++ Workshop, 1987.
- [5] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico: *Data Abstraction and Object-Oriented Programming in C++*. Wiley, 1990.
- [6] John A. Interrante, Mark A. Linton: *Runtime Access to Type Information in C++*. USENIX C++ Conference Proceedings, 1990.
- [7] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++*. USENIX C++ Conference Proceedings, 1990.
- [8] Mark A. Linton, John M. Vlissides, and Paul R. Calder: *Composing user interfaces with InterViews*. Computer, 22(2):8-22, February 1989.
- [9] Dmitry Lenkov, Michey Mehta, Shankar Unni: *Type Identification in C++*. USENIX C++ Conference Proceedings, April, 1991.
- [10] Bjarne Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley, 1991.
- [11] Bjarne Stroustrup and Dmitry Lenkov: *Run-Time Type Identification for C++*. Long version for ANSI/ISO committee discussions: ANSI/X3J16 document 92-00028. Shorter version: The C++ Report, Vol.4 No.3, pp 32-42. March/April 1992.
- [12] Andre Weinand, Erich Gamma, and Rudolf Marty: *ET++ - An Object-Oriented Application Framework in C++*. ACM OOPSLA'88 Conference Proceedings, 1988.

### 13 Appendix A: Declarations in Conditions

In §1 we mentioned in passing that we'd like to allow the use of declarations in conditions:

```
void my_fct(dialog_box* bp)
{
    if (dbox_w_str* dbp = (dbox_w_str*) bp) {

        // use 'dbp'

    }

    // ...
}
```

The value of a declaration is the value of the declared variable after initialization. To avoid syntax problems, we do not suggest that declarations can appear everywhere an expression can (which would be the cleanest semantic notion) but only that declarations of a single initialized variable can appear in the condition part of if, for, while, and switch statements. Allowing declarations in conditions of conditional expressions and do statements seems to add complications rather than utility so we don't propose that. For example:

```
do f() while(int i = g());           // error: declaration in do condition

while(int i = g()) f();              // ok

while(int i = g(), j = g2()) f();    // error: two names declared in condition
```

This extension is, of course, independent of the notion of run-time type identification. It simply attacks the problem of use of uninitialized variables directly. For example:

```
void f(Iter<Name> it)
{
    while (Record* r = it.next()) {
        // process '*r'
    }
}
```

The scope of a variable declared in a condition is the statement or statements controlled by the condition. In particular, a variable declared a condition of an if statement is in scope in the else part of that statement. Naturally, the variable will most often be 0 in the else statement, but it is possible to construct examples where it is not. For example, consider a class X with an operator int():

```
void g(double d)
{
    if (X x1 = d) {
        // we get here if x1.operator int()
        // doesn't yield 0
    }
    else {
        // x1 has a meaningful value even here
    }
}
```

It is not legal to declare a variable with the same name in both the condition and in the outermost block of a statement controlled by the condition. For example:

```
if (Name* p = find(s))
{
    char* p; // error: multiple definition of 'p'
    // ...
}
```

This rule parallels the rule that an argument name may not be redefined in the outermost block of a function:

```

void f(Name* p)
{
    char* p; // error: multiple definition of 'p'
    // ...
}

```

## 14 Appendix B: Casting from Virtual Bases

It is not possible to cast from a virtual base class to a derived class using an ordinary cast. This restriction does not apply to checked casts from polymorphic virtual base classes:

```

class B { /* ... */ virtual void f(); };
class V { /* ... */ virtual void g(); };

class D : public B, public virtual V { /* ... */ };

void g(D& d)
{
    B* pb = &d;
    D* pd1 = (D*)pb; // ok, unchecked
    D* pd2 = (?D*)pb; // ok, checked

    V* pv = &d;
    D* pd3 = (D*)pv; // error: cannot cast from virtual base
    D* pd4 = (?D*)pv; // ok, checked
}

```

The reason for the restriction to checked casts from polymorphic classes is that there isn't enough information available in other object to do the cast from a virtual base. In particular, an object of a type with layout constraints determined by some other language such as Fortran or C may be used as a virtual base class and for objects of such types only static type information will be available. However, the information needed to provide run time type identification includes the information needed to implement the checked cast.

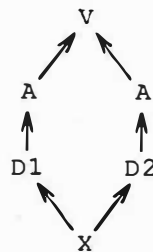
Naturally, such a cast can only be performed when it is unambiguous. Consider:

```

class A : public virtual V { /* ... */ };
class D1 : public A { /* ... */ };
class D2 : public A { /* ... */ };
class X : public D1, public D2 { /* ... */ };

```

Or graphically:



Here, an X object has two sub-objects of class A. Consequently, a cast from V to A within an X will be ambiguous and return a 0 rather than a pointer to an A:

```

void h1(X& x)
{
    V* pv = &x;
    A* pa = (?A*)pv; // pa will be initialized to 0
}

```

This ambiguity is not in general detectable at compile time:

```

void h2(V* pv)
{
    A* pa = (?A*)pv; // pv might point to an X
                    // and then 0 will be returned

                    // or it might point to a 'plain A'
                    // and then a correct pointer to A will be returned
}

```

This kind of run-time ambiguity detection is only needed for virtual bases. For ordinary bases, the proper sub-object to cast to can always be found; §8.

## 15 Appendix C: Explicit RTTI Declaration

Experience shows that providing checked casts for polymorphic types (only) works acceptably. However, people can get confused about which types are polymorphic. This leads to a wish for an explicit way of saying “this class supports RTTI whether it has virtual functions or not.”

First we note that there already is a way. Simply define a class with a virtual function and derive from it any class that you desire to be explicit about:

```

class rtti { virtual void __dummy() = 0; };
class X : public rtti { /* ... */ };

```

Unfortunately, this implies a space overhead (especially if `rtti` is included in lots of places) and because class `rtti` is so small, making it a virtual base will not provide any significant saving:

```

class X : public virtual rtti { /* ... */ };

```

It is also clear that “`public virtual rtti`” is long enough to be tedious to write so we considered some syntactic sugar:

```

class X : virtual { /* ... */ };
virtual class X { /* ... */ };
class X { virtual; /* ... */ };

```

However, people instantly started imagining a variety of meanings for such notations. In particular, “Oh neat, so `X` is an abstract class!” and “I have always wanted to be able to declare all functions `virtual` in one place” were not uncommon reactions. For now, we don’t have an acceptable suggestion for a more explicit way of saying “this class has run-time type information.” If you want such information, be sure to have at least one virtual function in the base class you want a checked cast from or leave it to the compiler to tell you.

## 16 Appendix D: Alternative `typeid()` Semantics

We considered two alternatives for the semantics of `typeid()`. Both are consistent and roughly equivalent. This appendix explains why we chose the one we did. Because we saw no major flaw in either alternative the discussion gets a bit involved.

To help the discussion, let’s here call the two alternatives `ptypoid()` and `otypoid()`. The `typeid()` semantics adopted and described in §6 is that of `otypoid()`. We will assume the definitions

```

class B { /* ... */ virtual void f(); }; // a polymorphic base class
class D : public B { /* ... */ };

B* p = new D; // a B* pointing to a D
B& r = *p;    // a B& referring to a D

```

### Semantics of `ptypoid()`

The original idea was `ptypoid(p)` meaning “the object describing the object pointed to by the pointer `p`.” One could imagine `ptypoid()` to take an argument of any pointer type:

```

template<class T> Type_info& ptypoid(T*);

```

The expected most common uses are:

```
if (ptypoid(p) == typeid(D)) // is the object pointed to by p a D?
if (ptypoid(&r) == typeid(D)) // is the object referred to by r a D?
```

Clearly the design of `ptypoid()` is geared to making the use of pointers convenient. What then if `p` is 0? This is easily handled by letting `typeid(0)` denote an object representing the 0 pointer; `typeid(0)` compares not equal to `typeid(T)` for every type `T`:

```
p = 0;
if (ptypoid(p) == typeid(D)) // fails
if (ptypoid(p) == typeid(0)) // succeeds
```

It is a compile time error to apply `ptypoid()` to a non-pointer:

```
int i;
ptypoid(i); // error: 'i' is not a pointer
ptypoid(7); // error: '7' is not a pointer
ptypoid(*p); // error: '*p' is not a pointer
```

So far, so good. Now consider references:

```
ptypoid(r); // == ptypoid(*p), that is, error, or
            // == typeid(D) ?

B& pr = p; // pr refers to a B*

ptypoid(pr); // == ptypoid(p) == typeid(D) or
             // == typeid(B*) or
             // == typeid(D) ?
```

In both cases, the first alternative is obtained by assuming that there is no special interaction between `ptypoid()` and references so that `ptypoid()` is applied to the object referred to, that is `*p` (of type `B`) a `p` (of type `B*`) respectively. The second alternative assumes that `ptypoid()` treats a reference similar to the way it treats a pointer, that is, it looks at the object referred to and finds its type. The third alternative for `ptypoid(pr)` is obtained by saying that both references and pointers are followed until a non-pointer and non-reference is found and that object is examined. The third alternative, “just chase pointers and references as far as we can” is an approach that has caused problems with other type systems and would be unique in C++ so we rejected that.

We considered the first two alternatives plausible. Using the first alternative, the result of `ptypoid(r)` will be surprising to many because it is the static type of the object referred to (`B`). Using the second alternative, the result of `ptypoid(pr)` will be surprising to many because `ptypoid(pr)` will differ from `ptypoid(p)` even though `pr` refers to `p`. In both cases comparisons with `typeid(D)` will fail.

Having `ptypoid(pr)` differ from `ptypoid(p)` even though `pr` refers to `p` seemed too odd a departure from the general rule that a name of an object and a reference to that object behave identically. Thus,

```
B& r = *p; // r refers to a D
B*& pr = p; // pr refers to a B*

ptypoid(r); // == ptypoid(*p), that is, error
ptypoid(pr); // == ptypoid(p) == typeid(D)
```

This means that to use `ptypoid()` effectively on a “typical reference argument” we must obtain a pointer using the address-of operator:

```
void f(B& r)
{
    if (ptypoid(&r) == typeid(D))
        // ...
}
```

This works nicely except where `&` is overloaded for `B`. It does look odd, though.

We conclude that `ptypoid()` can be made to work acceptably within the type system, but that there are a few details that are less than elegant.

### Semantics of `otypeid()`

Consider the `otypeid()` alternative: `otypeid(obj)` yields an object representing the type of `obj`. One could imagine `otypeid()` to take an argument of type "reference to any type:"

```
template<class T> const Type_info& otypeid(T&);
```

The expected most common uses are

```
if (otypeid(r) == typeid(D)) // is the object referred to by r a D?
if (otypeid(*p) == typeid(D)) // is the object pointed to by p a D?
```

Clearly the design of `otypeid()` is geared to making the use of references convenient. The `otypeid()` operator differs from the mythical `typeof()` operator only in that its call by reference semantics ensured that the dynamic type of an object is determined rather than its static type. For example, the (static) type of the expression `*p` is `B`, whereas `otypeid(*p)` is the type of the (dynamic) object pointed to by `p`, that is, `D`.

Applying `otypeid()` to non-pointers is no problem:

```
int i;
otypeid(i); // == typeid(int)
otypeid(7); // == typeid(int)
```

This implies that `otypeid(0)` isn't special:

```
p = 0;
otypeid(p); // == typeid(B*)
otypeid(0); // == typeid(int)
```

There is however, however, a problem related to "zero references:"

```
p = 0;
otypeid(*p); // == typeid(B) or
              // == typeid(void) or
              // throw exception ?
```

The first alternative simply returns the static type of `*p` if there is no object to examine. This is confusing and error prone. The second alternative returns a distinguished object (much as `ptypoid(0)` yields `typeid(0)`). The third alternative relies on the observation that any use of `*p` where `p==0` would be an error. The probable most common case would be the one where a reference `r` had somehow been bound to a non-object; that is, `&r==0`.

Any use of such an `r` will cause an error. The choice is between an explicit test to avoid such a use and the possibility of throwing an exception. Since one can already test for `&r==0` and `p==0` there is no need for an additional test `otypeid(*p)==typeid(void)`. Throwing an exception provides an implicit mechanism for detecting such errors. Therefore:

```
p = 0;
otypeid(*p); // throw exception
```

Again, the difference between pointers and references shows up in the way a reference to a non-existent object is handled.

### Comparison

Now consider a summary:



```
class B { /* ... */ virtual void f(); }; // a polymorphic base class
class D : public B { /* ... */ };
```

```
B* p = new D; // a B* pointing to a D
```

```
B& r = *p; // r refers to a D
B*& pr = p; // pr refers to a B*
```

ptypeid(p) == typeid(D)	otypeid(p) == typeid(B*)
ptypeid(*p) error	otypeid(*p) == typeid(D)
ptypeid(r) error,	otypeid(r) == typeid(D)
ptypeid(&r) == typeid(D)	otypeid(&r) == typeid(B*)
ptypeid(pr) == typeid(D)	otypeid(pr) == typeid(B*)
p = 0;	
ptypeid(p) == typeid(0)	otypeid(p) == typeid(B*)
ptypeid(*p) error	otypeid(*p) throw exception

If either `ptypeid()` or `otypeid()` should be called `typeid()` which would you choose? We chose `otypeid()` for several minor reasons.

We wanted a single `typeid()` operator that could be applied to both expressions and types. This weighed against the `ptypeid()` semantics. Having an implicit dereference for expression arguments but not for type arguments seemed odd:

```
typeid(p) == typeid(D) // but the type of p is B*, not D
```

This simple point becomes significant when thinking about templates. For example:

```
template<class T> const char* snameof(T& r) // return name of static type
{
    return typeid(T).name();
}

template<class T> const char* dnameof(T& r) // return name of dynamic type
{
    return typeid(r).name();
}
```

are simple to write using the `otypeid()` model.

Whatever we choose, somebody will make false assumptions about the model used for `typeid()` and write the equivalents to:

```
if (otypeid(p) == typeid(D)) // error or simply failed test?
// ...
if (ptypeid(*p) == typeid(D)) // error
```

The test will fail because `otypeid(p)` is `typeid(B*)`. However, this is a trap and might warrant a compiler warning. Making it an error to compare the `otypeid()` of a pointer to the `typeid()` of a non-pointer seems Draconian, though, and might complicate the writing of templates. Error handling is easier for the `ptypeid()` semantics so this favors the `ptypeid()` semantics.

If the address-of operator has been overloaded for a class then we cannot take the `ptypeid()` of a reference to that class:

```
class X {
    // ...
    X& operator&();
};

X x;
X& r = x;
ptypeid(&r); // error ptypeid() of non-pointer.
```

This could be a nasty problem. There is no equivalent problem for the `otypeid()` semantics because `*p`

is the application of a built-in operator to a pointer.

Finally, consider zero pointers. Using `ptypeid()` we would write

```
void f(B* p)
{
    if (ptypeid(p) == typeid(D) {
        // use p as a D*
    }

    // used p as a plain B*
}
```

and get burned if we use `p` as a plain `B*` without testing for 0. We could test for zero immediately upon entering `f()` or after the type test. Since `typeid(0)` is well defined it doesn't matter whether `p==0` is checked before or after.

Using `otypeid()` we would write

```
void f(B& r)
{
    if (otypeid(r) == typeid(D) {
        // use r as a D
    }

    // used r as a plain B
}
```

We are entitled to assume that `r` refers to an object and would normally not test for `&r==0`. If someone had cheated and passed a zero reference `otypeid(r)` will throw an exception. This is in our opinion preferable to the undefined behavior we'd get from using `r`. If we felt the need to check for `&r==0` we'd have to do it before using `otypeid(r)`:

```
void f(B& r)
{
    if (&r == 0) { // something is rotten
        // ...
    }
    else if (otypeid(r) == typeid(D) {
        // use r as a D
    }

    // used r as a plain B
}
```

We consider the behavior in the case of `p==0` very marginally in the favor of the `otypeid()` semantics.

Allowing `typeid(oo)` where the result will not depend on any run-time information – that is, where `oo` does not refer to a polymorphic type – could be considered redundant and therefore a possible source of confusion and errors. Instead of `typeid(oo)` you could use `typeid(T)` where `T` is the type of the object referred to by `oo`. This is an argument for the `ptypeid()` semantics. However, if you didn't declare `oo`, you don't necessarily know its type and whether that type is polymorphic or not. This can happen with templates with templates.

Further, suppose you're trying to define some kind of smart pointer class. If `typeid()` applies only to pointers, then it would be impossible to make `typeid()` work transparently with smart pointers. That is, if `p` is a smart pointer, `ptypeid(p)` would be illegal because `p` isn't what C++ thinks of as a pointer. However, `otypeid(*p)` would be whatever the type of `*p` is and `operator*()` can be defined for a smart pointer type.

## 17 Appendix E: Return Types

The March 1992 meeting of the ANSI/ISO C++ standards committee in London decided – after almost two years of deliberations – to relax the rules for overriding virtual functions to allow a function returning a B\* to be overridden by a function returning a D\* when B is a public base class of D. Similarly, a function returning a B& can be overridden by a function returning a D&.

This provides an alternative to uses of casts that might have been considered candidates for RTTI. For example,

```
class X {
    // ...
    virtual X* clone(); // return copy of *this
};

class Y : public X {
    // ...
    X* clone();
}

void f(X* p, Y* q)
{
    X* pp = p->clone();
    Y* qq = (Y*) q->clone(); // the clone of a Y is at least a Y

    if (Y* q2 = (?Y*) pp) { // was the X really a Y?
        // ...
    }
    // ...
}
```

The return type relaxation makes a better solution possible:

```
class Y : public X {
    // ...
    Y* clone(); // override X::clone()
}

void f(X* p, Y* q)
{
    X* pp = p->clone();
    Y* qq = q->clone(); //no cast (checked or unchecked) needed

    if (Y* q2 = (?Y*) pp) { // was the X really a Y?
        // ...
    }
    // ...
}
```

We mention this to remind people that blindly changing all casts to checked casts isn't a good way to try to improve old programs.

## 18 Appendix F: typeid() During Construction

A call of typeid(\*p) reflects the dynamic type of \*p in the same way as a call of a virtual function p->f() would. This implies that in a constructor or destructor of X the call typeid(\*this) will be return(X) rather than the typeid() of some derived class that the X object might be part of after construction and before destruction.



# Run-Time Type Information and Class Design

Annotations to Stroustrup & Lenkov

Doug Lea

Computer Science, SUNY Oswego, Oswego NY 13126

and

New York CASE Center, Syracuse University, Syracuse NY 13244

dl@g.oswego.edu

## Abstract

Design applications of the run-time type identification constructs proposed by Stroustrup and Lenkov are illustrated via several examples that demonstrate their strengths and weaknesses as tools in object-oriented design.

Some people think that run-time type identification (RTTI) constructs cause programmers to sidestep many of the good design practices evident in well-crafted object-oriented programs. Others think that it is impossible to even write well-crafted object-oriented programs without run-time type support. This commentary paper briefly attempts to disentangle some of the issues behind such views. A few problems are illustrated for which RTTI might plausibly be used to help formulate a solution. These lead to a discussion of some underlying design and engineering considerations, and allow some tentative conclusions (noted within boxes throughout the paper).

## Example 1

The first example involves probably the most common application of RTTI. Assume a base class, along with a subclass that possesses additional properties not listed in the base. For example:

```
class Person { ... };
class Employee : public Person {
public:
    virtual float salary() const;
    virtual Department* dept();
    ...
};
```

Along with a heterogeneous collection class, for example:

```
class PersonList {
public:
    Person* first();
    Person* next(Person*);
    ...
};
```

And finally the problem:

Write `sumSalaries(PersonList* l)`, that returns the sum of all salaries in `l`.

This is an impossible demand, since `PersonList` entries don't necessarily possess `salary` attributes. At best, we can sum the salaries for all (sub)Persons known to contain a `salary`. In doing so, we might arbitrarily decide to treat all others as having a salary of zero. Given this, a solution may be had using RTTI:

```
float sumSalaries(PersonList* l)
{
    float sum = 0.0;
    for (Person* p = l->first(); p != 0; p = l->next(p))
        if (Employee* e = (Employee*)p) sum += e->salary();
    return sum;
}
```

*RTTI can make heterogeneous collections more usable in C++.*

## Post Hoc Attributes

The `sumSalaries` procedure might be seen as implicitly attaching a *new* property to class `Person`, namely:

```
class Person { ...
    virtual float salary() const { return 0.0; }
};
```

The fact that this was done *implicitly* seems innocent enough. But what if some other procedure having to do with salaries and persons made a different decision; e.g., that unless specified, the salary of a `Person` should be estimated as the average yearly per capita income? This is the sort of software management problem that classes, encapsulation, and inheritance were meant to *solve*, not *create*. And this is the sort of usage that gives RTTI a bad reputation.

It would have been much better to build the default `salary` attribute into `Person` to begin with. But perhaps the class "belongs" to someone else and cannot be changed. Perhaps changes would break other existing code. There are many such reasons for not touching class interfaces when you don't absolutely have to. There is a nicer-looking solution. It may be approached through a version that looks even nicer still, but does not work as naively expected:

```
float getSalary(Person* p) { return 0.0; } // wrong
float getSalary(Employee* p) { return p->salary(); }

float sumSalaries_2(PersonList* l)
{
    float sum = 0.0;
    for (Person* p = l->first(); p != 0; p = l->next(p))
        sum += getSalary(p);
    return sum;
}
```

This is better than the original version, since the decision to treat non-existing salaries as zero is clearly enshrined within independent procedures that all other classes and procedures may use.

Unfortunately, the code does not solve the problem. C++ does not dynamically dispatch procedures on the basis of *arguments*, only *receivers*<sup>1</sup>. Thus `sumSalaries_2` would always return zero. However, *this* can be fixed using RTTI:

```
float getSalary(Person* p)
{
    Employee* e;
    if (e = (Employee*)p) return e->salary(); else return 0.0;
}
```

*RTTI can simulate and implement dynamic argument-based dispatching.*

<sup>1</sup> and only when declared virtual, etc.

## Class tests and feature tests

All is well with the above solution until the day someone adds:

```
class Contractor : public Person { ...  
    virtual float salary() const;  
    virtual Job* job();  
};
```

Contractors aren't Employees, yet they also have salary attributes. If a Contractor ever shows up in a PersonList, then both sumSalaries and sumSalaries\_2 will treat its salary as zero. This is probably not what anyone had in mind.

The problem is that the *class name* Employee was an alias for possession of the *property* (method) salary. This trick works only when it works.

*RTTI cannot be used to infer features unless classes have been designed to support this to begin with.*

In the current example there are several cures, including:

1. Finally add salary to Person.
2. Add subclass SalariedPerson: public Person and adjust the Employee and Contractor class declarations accordingly.
3. Add class Salaried as a "mixin" class, and adjust Employee and Contractor classes to multiply inherit both Salaried and Person.
4. Rewrite getSalary to investigate possession of salary through Type\_info information rather than through the conditional cast mechanism.

Each of these has its merits. Each also requires changes to existing code after the introduction of Contractor. RTTI does not always eliminate the need for such alterations.

*RTTI can postpone necessary refactorings.*

Note however that any of these strategies *could* have been applied in our original versions. People tend not to do so though. Routine creation of extremely fine-grained classes corresponding to *each* "added" property gets pretty tedious, as does the alternative of routinely extracting Type\_info information probing for possession of these properties. These human-factors considerations are sometimes serious barriers to extensibility. RTTI offers an incomplete solution. (Other equally incomplete solutions include *views* [5] and *conformance* based typing [4].)

## Example 2

This example was made famous in a set of Usenet postings:

```
class Driver { ... };  
class ProDriver : public Driver { ... };  
  
class Vehicle { ...  
    virtual void Register(Driver* d) { vd(); }  
};  
  
class Truck : public Vehicle { ...  
    void Register(ProDriver* d) { tp(); }  
};
```

The *idea* here seems to be that a `Vehicle` may be registered to any kind of `Driver`, but a `Truck` may only be registered (in some perhaps different way, as signified by `tp()` vs `vd()`) to a `ProDriver` (professional driver).

The above declarations are not quite illegal C++<sup>2</sup> but do not work as expected. For example,

```
void reg(Vehicle* v, Driver* d) { v.Register(d); }

main() {
    Truck* t = new Truck;
    ProDriver* p = new ProDriver;
    t.Register(p);          // Truck::Register(ProDriver*) invoked
    reg(t, p);              // Vehicle::Register(Driver*) invoked via reg
}
```

This is a more subtle consequence of C++ rules that dynamically dispatch only on receiver, not argument types.

## Multimethods

A cure may be obtained by “pulling out” the `Register` method from the classes and using RTTI.

```
void Register(Vehicle* v, Driver* d) {
    if ((Truck*)(v) && (ProDriver*)(p)) tp(); else vd();
}
```

This style of specialization based on the types of (potentially) *ALL* participants in an operation is called *multimethod dispatching*. CLOS [1] is justly famous for supporting multimethods as first-class programming constructs.<sup>3</sup> If C++ supported multimethods directly, then this might have been written somewhat more clearly and extensibly:

```
void Register(Vehicle* v, Driver* d) { vd(); }

void Register(Truck* v, ProDriver* d) { tp(); }
```

*RTTI can simulate and implement multimethods.*

## Types as Guards

The simulated multimethod solution has the advantage of predictable dispatching. This is vital in order to statically determine correctness, or even reasonableness. It's hard to say very much at all about the original version. In practice, using RTTI-simulated multimethods to control dispatching of special cases of overloaded methods and procedures is much safer and more reliable than depending on C++ “overload resolution” policies.

But in the current example, the improved clarity highlights *conceptual* problems with the design. The probable intent was to *disallow* all but `ProDrivers` from registering `Trucks`. The above solution allows `Drivers` to register them, but uses the `vd()` code in `Vehicle::Register` to do so. The use of multimethod dispatch seems like the wrong way to address this. It uses type information to *direct*, not *guard* or prohibit certain calls.

In these kinds of designs, there is simply no way to statically prohibit certain argument combinations. The special cases must be considered truly *exceptional* to the general `Vehicle-Driver` relationship. Probably the best solution here would be to explicitly indicate possible failure to clients. This could be done in several ways, including:

<sup>2</sup> See [2] chapter 13 for the gruesome details.

<sup>3</sup> The remarks in [6] about specifically not including multimethods in their proposal seem misplaced given that many *uses* of RTTI amount to their simulation. The main difference and advantage of first-class multimethods is that they are extensible – new special cases may be added without modifying existing code. In any case, multimethods and RTTI can each fully simulate the other.



```

bool Register(Vehicle* v, Driver* d) {
    if ((Truck*)(v))
        if ((ProDriver*)(p)) { tp(); return TRUE; }
        else return FALSE;
    else { vd(); return TRUE; }
}

```

*RTTI can detect exceptional argument combinations that cannot be statically prohibited.*

## Example 3

Suppose we are building a class representing face icons that may be in any of three states, happy, sad, and asleep. One design strategy is to create three different classes, one per state and a “controller”, that switches among them. This is an attractive *delegation* [3] based design:

```

class IconState {
    virtual bool eyesOpen() const = 0;
    ...
};

class HappyIcon : public IconState {
    bool eyesOpen() const { return TRUE; }
    ...
};

class AsleepIcon : public IconState { ... }
class SadIcon : public IconState { ... }

class Icon {
    IconState* theIcon;

    virtual bool eyesOpen const { return theIcon->eyesOpen(); }
    ...
    virtual bool isHappy const { return ((HappyIcon*)(theIcon) != 0); }
};

```

This is a situation in which RTTI is clearly the *best* alternative. How else would an Icon know which state it were in within `isHappy`? Alternatives like maintaining logical variables invite needless error-prone complexity.

Importantly, this strategy extends to testing and internal integrity checking. For example, the Icon class requires a `beHappy` method to change state:

```

...
virtual void beHappy() {
    theIcon = getHappyIcon();
    assert(isHappy());
}

```

The main reason this works so nicely here is that we have carefully merged the notions of class membership and property (and/or property value) possession. This takes some planning.

*RTTI can be used to prescribe, determine, and verify logical state.*

## Example 4

Suppose we need to design a long-lived application program with fault-tolerance support in case of crashes. We settle on a checkpoint/rollback scheme in which the states of all objects are periodically

saved on disk. Recovery is performed by re-constructing or reinitializing (depending on the nature of the crash) all objects to their last saved states.

Design and implementation of such mechanisms is not an easy matter. Doing a thorough job is tantamount to the construction of an object-oriented database system. But there is the widespread belief that RTTI substantially simplifies practical application-specific solutions.

“Substantially” is much too strong a term here. Most of the snags in this kind of persistence support revolve around the transformation and associated bookkeeping of *object* identities, that are internally represented through pointer values, but externally through some other scheme (e.g., integer pseudo-identities). *Class* identities must also be stored and recovered in order to allow reconstruction. RTTI *per se* can only assist in only the latter.

*RTTI does not automate persistence support.*

## Save / Restore Mechanics

On the other hand, RTTI certainly doesn't make this any *harder*. There are many ways to design a save/restore mechanism. Here is a simplified prototypical framework<sup>4</sup>. On the save side, for each object to be stored:

1. Output an external pseudo-identity uniquely corresponding to its internal ID (address)
2. Output an external pseudo-identity uniquely corresponding to its maximal internal class ID.
3. Output all values of “simple” state attributes of built-in type.
4. Output all pseudo-IDs corresponding to pointer attributes.
5. Ensure that at some point values and pseudo-IDs for all **static** class data for this object's class are saved.

The restore side is mostly symmetrical. For each object to be recovered:

1. Read in an external object pseudo-ID. Map it either to an existing internal ID or to one that is to be constructed (depending on the kind of recovery).
2. Read in an external class pseudo-ID. Use it as a key to dispatch to a routine that constructs or re-initializes the object using the value and pointer information to follow. In other words, the “driver” routine is a big **switch** statement, although perhaps a well-disguised one. It may also need to queue or reorder requests in order to delay the construction of objects until their components exist.
3. Somehow separately handle **static** class data.

## Metaclasses

There are a number of ways in which RTTI can make these tasks a bit easier to implement, without otherwise affecting their logic one way or the other. These mainly arise through exploitation of extensible `Type_info` structures.

- `typeid` values could be used as the internal class pseudo-IDs. This simplifies dispatch logic in the recovery routine.

---

<sup>4</sup>For example, among the simplifications is that it does not accommodate “embedded” objects; i.e., those that directly nest one object within another.

- In particular, access to the (re)initialization routines could be arranged through `Type_info` structures indexed by the `typeid`.
- Maps between internal and external IDs could be located in per-class `Type_info` structures.
- Bookkeeping on recovery of static data could be located in `Type_info` structures.

It should soon occur to anyone familiar with languages like Smalltalk that the logic of grouping these kinds of per-class bookkeeping routines in a central place leads to the notion of *metaclasses* as a replacement for C++-style `static` class functions including, significantly, client-accessible *constructors*. This may in turn lead to a very different style of class design in general – for many purposes, `typeid(p).info()` might as well be pronounced “p’s metaclass”.

*RTTI and extensible `Type_info` classes can simulate metaclasses.*

## References

- [1] Bobrow, D., L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, & D. Moon, “Common Lisp Object System”, *SIGPLAN Notices*, September, 1988.
- [2] Ellis, M., & B. Stroustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
- [3] Johnson, R., & J. Zweig. “Delegation in C++”, *Journal of Object-Oriented Programming*, November, 1991.
- [4] Russo, V. & E. Granston, “Signature-based polymorphism for C++” *Proceedings, 1991 Usenix C++ Conference*, Washington, 1991.
- [5] Scholl, M, C. Laasch, & M. Tresch, “Updatable views in object-oriented databases”, in C. Delobel, M. Kifer & Y. Masunaga (eds.) *Deductive and Object-Oriented Databases*, Springer-Verlag, 1991.
- [6] Stroustrup, B., & D. Lenkov, “Run-time type identification for C++”. *Proceedings, 1992 Usenix C++ Conference*, Portland, 1992.



## The USENIX Association

USENIX, the UNIX and Advanced Computing Systems professional and technical organization, is a not-for-profit membership association made up of systems researchers and developers, systems administrators, programmers, support staff, application developers and educators.

USENIX is dedicated to:

- \* fostering innovation and communicating research and technological developments,
- \* sharing ideas and experience, relevant to UNIX, UNIX-related and advanced computing systems
- \* providing a forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, the Association sponsors two annual technical conferences and frequent symposia and workshops addressing special interest topics, such as C++, distributed systems, Mach, systems administration, and security. USENIX publishes proceedings of its meetings, a bi-monthly newsletter *login.*, and a refereed technical quarterly, *Computing Systems*. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

There are four classes of membership in the Association: student, individual, institutional, and supporting, differentiated primarily by the fees paid and services provided. The supporting members of the Association are:

Digital Equipment Corporation  
Frame Technology, Inc.  
Matsushita Graphic Communication Systems, Inc.  
mt Xinu  
Open Software Foundation  
Quality Micro Systems  
Rational Corporation  
Sun Microsystems, Inc.  
Sybase, Inc.  
UNIX System Laboratories, Inc.  
UUNET Technologies, Inc.

For further information about membership or to order publications, contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710-2565  
Telephone: 510/528-8649  
Email: [office@usenix.org](mailto:office@usenix.org)  
Fax: 510/548-5738

ISBN 1-880446-45-6